Data Structures and Algorithms

Topic 9

Lists

- Why lists?
- List windows
- Specification
- Block representation
- Singly linked representation
- Performance comparisons

Reading: Wood, Secs. 3.1, 3.3, 3.4

1. Introduction

Queues and stacks are restrictive — can only access one position ("first" in queue, "top" of stack)

In some applications we want to access sequence at many different positions:

eg. Text editor — sequence of characters, read/insert/delete at any point

eg. Bibliography — sequence of bibliographic entries

eg. Manipulation of polynomials — see Wood, Sec. 3.3.

eg. List of addresses

⋮

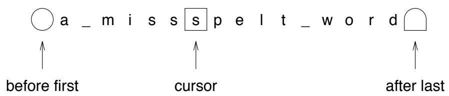
In this section we introduce the List ADT which generalises stacks and queues.

2. List windows

We will use the word "window" to refer to a specific position in the list:

- maintain a distinction from "reference" or "index" which are specific implementations
- maintain a distinction from "cursor" which is most commonly used as an application of a window in editing

May be several windows, eg. . .



Our List ADT will provide explicit operations for handling windows.

The following specification assumes that *w* is a Window object, defined in a separate class. Different window objects will be needed for different List implementations

⇒ a List class and a companion Window class will be developed together. . .

3. List Specification

☐ **Constructors**

1. *List()*: Initialises an empty list with two associated window positions, *before first* and *after last*.

☐ **Checkers**

2. *isEmpty()*: Returns *true* if the list is empty.
3. *isBeforeFirst(w)*: True if *w* is over the before first position.
4. *isAfterLast(w)*: True if *w* is over the after last position.

☐ **Manipulators**

5. *beforeFirst(w)*: Initialises *w* to the before first position.
6. *afterLast(w)*: Initialises *w* to the after last position.
7. *next(w)*: Throws an exception if *w* is over the after last position, otherwise moves *w* to the next window position.

8. *previous(w)*: Throws an exception if *w* over is the before first position, otherwise moves *w* to the previous window position.
9. *insertAfter(e,w)*: Throws an exception if *w* is over the after last position, otherwise an extra element *e* is added to the list after *w*.
10. *insertBefore(e,w)*: Throws an exception if *w* is over the before first position, otherwise an extra element *e* is added to the list before *w*.
11. *examine(w)*: Throws an exception if *w* is in the before first or after last position, otherwise returns the element under *w*.
12. *replace(e,w)*: Throws an exception if *w* is in the before first or after last position, otherwise replaces the element under *w* with *e* and returns the old element.
13. *delete(w)*: Throws an exception if *w* is in the before first or after last position, otherwise deletes and returns the element under *w*, and places *w* over the next element.

3.1 Simplifying Assumptions

Allowing multiple windows can introduce problems. Consider the following use of the List ADT:

```
    .
    .
Window w1 = new Window();
Window w2 = new Window();

beforeFirst(w1);            {Initialise first window.}
next(w1);                   {Place over first element.}
beforeFirst(w2);            {Initialise second window.}
next(w2);                   {Place over first element.}
delete(w1);                 {Delete first element.}
```

Our spec doesn't say what happens to the second window!

Number of options, e.g....

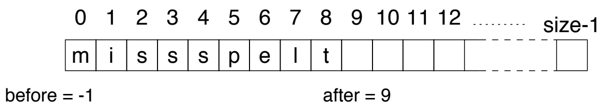
- other windows become undefined
- other windows treated in same way as first

We will not worry about details here.

4. Block Representation

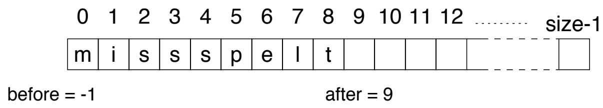
List is defined on a block (array)...

```
public class ListBlock {

  private Object[] block;    \\ holds general objects
  private int before;        \\ index to before first position
  private int after;         \\ index to after last position
```



Constructor

```
public ListBlock (int size) {
  block = new Object[size];
  before = -1;
  after = 0;
}
```



Windows

Some ADTs we have created have implicit windows — eg Queue has a "window" to the first item.

There was a fixed number of these, and they were built into the ADT implementation — eg a member variable `first` held an index to the block holding the queue.

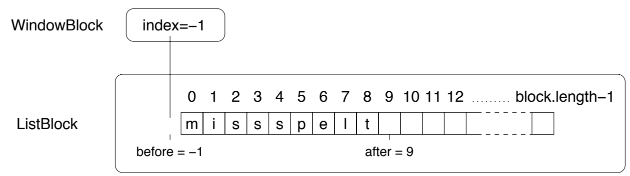
For List the user needs to be able to create arbitrarily many windows $\Rightarrow$ we define these as separate objects.

For the block representation, they just hold an index...

```
public class WindowBlock {
  public int index;
  public WindowBlock () {}
}
```

The index is then initialised by a call to *beforeFirst* or *afterLast*.

```
public void beforeFirst (WindowBlock w) {w.index = before;}
```



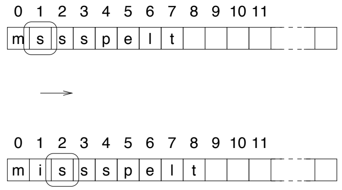
`next` and `previous` simply increment or decrement the window position. . .

```
public void next (WindowBlock w) throws OutOfBounds {
  if (!isAfterLast(w))  w.index++;
  else
    throw new OutOfBounds("Calling next after list end.");
}
```

`examine` and `replace` are simple array assignments.

Insertion and deletion may require moving many elements
⇒ worst-case performance — *linear* in size of block

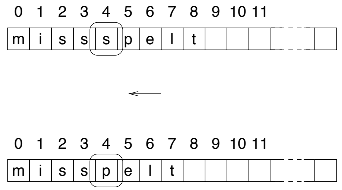
eg. `insertBefore`



From an ‘abstract’ point of view, window hasn’t moved — still over same element. However the ‘physical’ location has changed.

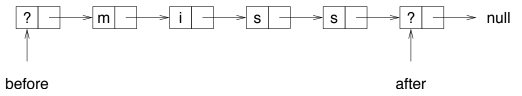

```
public void insertBefore (Object e, WindowBlock w) throws
OutOfBounds, Overflow {
  if (!isFull()) {
    if (!isBeforeFirst(w)) {
      for (int i = after-1; i >= w.index; i--)
        block[i+1] = block[i];
      after++;
      block[w.index] = e;
      w.index++;
    }
    else throw new OutOfBounds ("Inserting before start.");
  }
  else throw new Overflow("Inserting in full list.");
}
```


eg. `delete`



Window has moved from an ‘abstract’ point of view, tho ‘physical’ location is the same.

5. Singly Linked Representation



Uses two *sentinel* cells for before first and after last:

- *previous* and *next* always well-defined, even from first or last element
- Constant time implementation for *beforeFirst* and *afterLast*

Empty list just has two sentinel cells. . .


```
public class ListLinked {

  private Link before;
  private Link after;

  public ListLinked () {
    after = new Link(null, null);
    before = new Link(null, after);
  }

  public boolean isEmpty () {return before.successor == after;}
```


Windows

```
public class WindowLinked {
  public Link link;
  public WindowLinked () {link = null;}
}
```

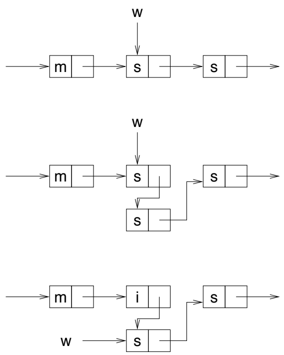
eg.

```
  public void beforeFirst (WindowLinked w) {w.link = before;}

  public void next (WindowLinked w) throws OutOfBounds {
    if (!isAfterLast(w))  w.link = w.link.successor;
    else
      throw new OutOfBounds("Calling next after list end.");
  }
```


`insertBefore` and `delete`

Problem — need *previous* cell! To find this takes linear rather than constant time.

One solution: insert *after* and swap items around



```
public void insertBefore (Object e, WindowLinked w) throws
OutOfBounds {
  if (!isBeforeFirst(w)) {
    w.link.successor = new Link(w.link.item, w.link.successor);
    if (isAfterLast(w))  after = w.link.successor;
    w.link.item = e;
    w.link = w.link.successor;
  }
  else throw new OutOfBounds ("inserting before start of list");
}
```

Alternative solution: define window value to be the link to the cell previous to the cell in the window — Exercise.

5.1 Implementing previous

To find the previous element in a singly linked list we must start at the first sentinel cell and traverse the list to the current window, while storing the previous position. . .

```
public void previous (WindowLinked w) throws
OutOfBounds {
  if (!isBeforeFirst(w)) {
    Link current = before.successor;
    Link previous = before;
    while (current != w.link) {
      previous = current;
      current = current.successor;
    }
    w.link = previous;
  }
  else throw new OutOfBounds ("Calling previous before start of list.");
}
```


This is called *link coupling* — linear in size of list!

Note: We have assumed (as in previous methods) that the window passed is a valid window to *this* List.

In this case if this is not true, Java will throw an exception when the `while` loop reaches the end of the list.

6. Performance Comparisons

Operation	Block	Singly linked
List	1	1
isEmpty	1	1
isBeforeFirst	1	1
isAfterLast	1	1
beforeFirst	1	1
afterLast	1	1
next	1	1
previous	1	n
insertAfter	n	1
insertBefore	n	1
examine	1	1
replace	1	1
delete	n	1

In addition to *fixed maximum length*, block representation takes *linear time* *for insertions and deletions.*

Singly linked wins on all accounts except *previous*, which we address in the next section!

7. Summary

- Lists generalise stacks and queues by enabling insertion, examination and deletion at any point in the sequence.
- Insertion, examination and deletion are achieved using *windows* on the list.
- Explicit window manipulation is included in the specification of our List ADT.
- Block representation restricts list size and gives linear time results for insertions and deletions.
- Singly linked representation allows arbitrary size lists, and is constant time in all operations except *previous*.

Data Structures and Algorithms

Topic 10

Simplists and other List Variations

- More on the List ADT
  - doubly linked lists
  - circularly linked lists
  - performance
- The Simplist ADT
  - specification
  - singly linked type declaration
  - reference reversal
  - amortized analysis
  - performance comparisons

Reading

Wood, Chapter 3, Section 3.4.2 onwards

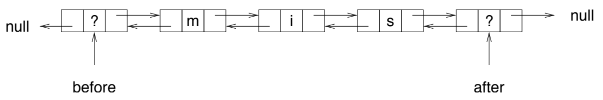
1. Doubly Linked Lists

Singly linked list:

- arbitrary size
- constant time in all operations, except *previous*

previous linear time in worst case — may have to search through whole list to find previous window position.

One solution — keep references in both directions!



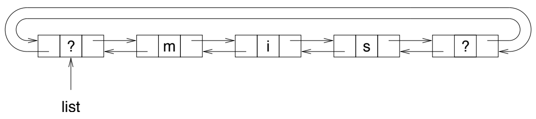
Called a *doubly linked list*.

Advantage *previous* is similar to *next* — easy to program and constant time.

Disadvantage Extra storage required in each cell, more references to update.

2. Circularly Linked Lists

The doubly linked list has two wasted pointers. If we link these round to the other end. . .



Called a *circularly linked list*.

Advantages (over doubly linked)

- Only need a reference to the first sentinel cell.
- Elegant!

Redefine Link

```
public class LinkDouble {
  public Object item;
  public LinkDouble successor;
  public LinkDouble predecessor;         // extra cell
```

Redefine List

```
public class ListLinkedCircular {
  private LinkDouble list;              // just one reference
```

Code for previous

```
public void previous (WindowLinked w) throws
OutOfBounds {
  if (!isBeforeFirst(w))  w.link = w.link.predecessor;
  else throw
    new OutOfBounds("calling previous before start of list ");
}
```

Cf. previous *previous*!

3. Performance — List

Operation	Block	Singly linked	Doubly linked
List	1	1	1
isEmpty	1	1	1
isBeforeFirst	1	1	1
isAfterLast	1	1	1
beforeFirst	1	1	1
afterLast	1	1	1
next	1	1	1
previous	1	n	1
insertAfter	n	1	1
insertBefore	n	1	1
examine	1	1	1
replace	1	1	1
delete	n	1	1

We see that doubly linked has superior performance. This needs to be

weighed against additional space overheads.

Rough rule

- *previous* commonly used $\Rightarrow$ doubly (circularly) linked
- *previous* never or rarely used $\Rightarrow$ singly linked

4. The Simplist ADT

The List ADT provides multiple explicit windows — we need to identify and manipulate windows in any program which uses the code.

If we only need a single window (eg a simple "cursor" editor) we can write a simpler ADT $\Rightarrow$ Simplist.

- single, implicit window (like Queue or Stack) — no need for arguments in the procedures to refer to the window position

We'll also provide only one window initialisation operation, *beforeFirst*

We'll show that, because of the single window, all ops except *beforeFirst* can be implemented in constant time using a singly linked list! Uses a technique called *pointer reversal* (or *reference reversal*).

We also give a useful amortized result for *beforeFirst* which shows it will not be too expensive over a collection of operations.

4.1 Simplist Specification

□ **Constructor**

1. *Simplist()*: Creates an empty list with two window positions, before first and after last, and the window over before first.

□ **Checkers**

2. *isEmpty()*: Returns true if the simplist is empty.
3. *isBeforeFirst()*: True if the window is over the before first position.
4. *isAfterLast()*: True if the window is over the after last position.

□ **Manipulators**

5. *beforeFirst()*: Initialises the window to be the before first position.
6. *next()*: Throws an exception if the window is over the after last position, otherwise the window is moved to the next position.
7. *previous()*: Throws an exception if the window is over the before first position, otherwise the window is moved to the previous position.

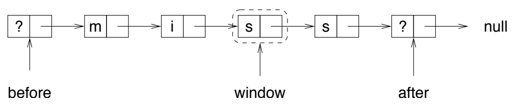
8. *insertAfter(e)*: Throws an exception if the window is over the after last position, otherwise an extra element *e* is added to the simplist after the window position.
9. *insertBefore(e)*: Throws an exception if the window is over the before first position, otherwise an extra element *e* is added to the simplist before the window position.
10. *examine()*: Throws an exception if the window is over the before first or after last positions, otherwise returns the value of the element under the window.
11. *replace(e)*: Throws an exception if the window is over the before first or after last positions, otherwise replaces the element under the window with *e* and returns the replaced element.
12. *delete()*: Throws an exception if the window is over the before first or after last positions, otherwised the element under the window is removed and returned, and the window is moved to the following position.

4.2 Singly Linked Representation

Again block, doubly linked are possible — same advantages/disadvantages as List. Our aim is to show an improvement in singly linked.

Since the window position is not passed as an argument, we need to store it in the data structure...

```
public class SimplistLinked {

  private Link before;
  private Link after;
  private Link window;
```



4.3 Reference (or "Pointer") Reversal

The window starts at *before first* and can move up and down the list using *next* and *previous*.

Problem

As for singly linked List, *previous* can be found by link coupling, but this takes linear time.

Solution

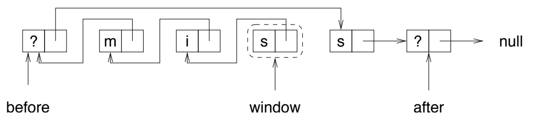
Q: What do you always do when you walk into a labyrinth?

Solution...

- point successor fields behind you backwards
- point successor fields in front of you forwards

Problem: window cell can only point one way.

Solution: before first successor no longer needs to reference first element (can follow references back). Use it to reference cell after window, and point window cell backwards.



⇒ *reference (pointer) reversal*

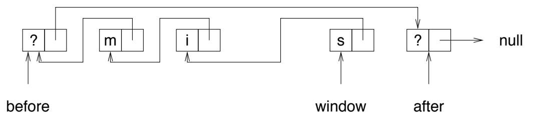
Exercise

```
public void previous() {
  if (!isBeforeFirst) {





  }
  else throw
    new OutOfBounds("calling previous before start of list");
}
```

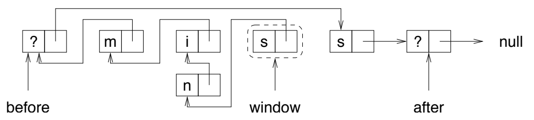
What is the performance of *previous*?

Other operations also require reference reversal.

delete. . .



insertBefore. . .



Disadvantage(?): A little more complex to code.

Advantage: Doesn't require extra space overheads of doubly linked list.

A outweighs D — you only code once, might use many times!

Problem: These ops only reverse one or two references, but what about *beforeFirst*? Must reverse references back to the beginning. (Note that *previous* and *next* now modify list *structure*.)

⇒ linear in worst case

What about amortized case. . .

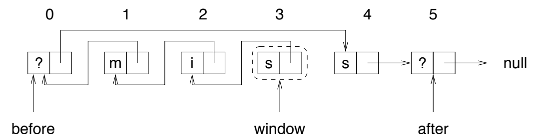
4.4 Amortized Analysis

Consider the operation of the window prior to any call to *beforeFirst* (other than the first one).

Must have started at the before first position after last call to *beforeFirst*.

Can only have moved forward by calls to *next* and *insertBefore*.

If window is over the *i*th cell (numbering from 0 at before first), there must have been i calls to *next* and *insertBefore*. Each is constant time, say 1 "unit".



beforeFirst requires i constant time "operations" (reversal of i pointers) — takes i time "units".

Total time: $2i$. Total number of ops: $i+1$.

Average time per op: ≈ 2

Average time over a sequence of ops is (roughly) constant!

Formally: *Each sequence of n operations takes $O(n)$ time; ie each operation takes constant time in the amortized case.*

4.5 Performance Comparisons — Simplist

Operation	Block	Singly linked	Doubly linked
Simplist	1	1	1
isEmpty	1	1	1
isBeforeFirst	1	1	1
isAfterLast	1	1	1
beforeFirst	1	1^a	1
next	1	1	1
previous	1	1	1
insertAfter	n	1	1
insertBefore	n	1	1
examine	1	1	1
replace	1	1	1
delete	n	1	1

a — amortized bound

5. Summary

Lists

- Block
  - bounded
  - linear time insertions and deletions, other ops constant
- Singly Linked
  - linear only for *previous*, other ops constant
- Doubly (and Circularly) Linked
  - constant time performance on all operations
  - needs extra space

Simplists

- Block, Doubly and Circularly Linked
  - as above
- Singly Linked with Reference Reversal
  - constant amortized case performance in all operations

Data Structures and Algorithms

Topic 11

Maps and Binary Search

- Definitions — what is a map (or function)?
- Why study maps?
- Specification
- List-based representation (singly linked)
- Sorted block representation
  - binary search, performance of binary search
- Performance comparison

Reading

Wood, Section 4.1, 4.2, 4.5

1. What is a map (or function)?

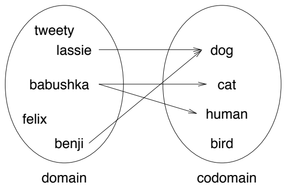
Some definitions...

relation — set of n-tuples

eg. $\{\langle 1, i, a\rangle, \langle 2, ii, b\rangle, \langle 3, iii, c\rangle, \langle 4, iv, d\rangle, \ldots\}$

binary relation — set of pairs (2-tuples)

eg. $\{\langle lassie, dog\rangle, \langle babushka, cat\rangle, \langle benji, dog\rangle, \langle babushka, human\rangle, \ldots\}$



dog is called the *image* of *lassie* under the relation

domain — set of values which can be taken by first item of a binary relation

eg. $\{lassie, babushka, benji, felix, tweety\}$

codomain — set of values which can be taken by second item of a binary relation

eg. $\{dog, cat, human, bird\}$

map (or function) — binary relation in which each element in the domain is mapped to *at most one* element in the codomain (*many-to-one*)

eg.

$$\text{affiliation} = \{ \begin{array}{l} \langle\ \text{chamarette}\ ,\ \text{green}\ \rangle \\ \langle\ \text{lawrence}\ ,\ \text{labour}\ \rangle \\ \langle\ \text{howard}\ ,\ \text{liberal}\ \rangle \\ \langle\ \text{kernot}\ ,\ \text{democrat}\ \rangle \\ \langle\ \text{smith}\ ,\ \text{labour}\ \rangle \\ \langle\ \text{kailis}\ ,\ \text{natural law}\ \rangle \} \end{array}$$

Shorthand notation: eg. affiliation(chamarette) = green

partial map — not every element of the domain has an image under the map (ie, the image is undefined for some elements)

2. Aside: Why Study Maps?

A Java method is a function or map — why implement our own map as an ADT?

- Create, modify, and delete maps during use.

  eg. a map of party affiliations may change over time — Rocher was Liberal for one term, Independent for the next

  A Java program cannot modify itself (and therefore its methods) during execution (some languages, eg Prolog, can!)

[Aside: a *meta-level* language can be used to *talk about* an *object level* language

eg. if I describe some mathematical equations, English is the meta-language; the numbers and $+$, $\times$, $\log$, etc form the object language

eg. if I write a program to manipulate data in a specified format, that format describes the object language, and the programming language acts as a meta-language

We want our functions to be described at the *object* level rather than the *meta*-level.]

- Java methods just return a result — we want more functionality (eg. ask "is the map defined for a particular domain element?")

3. MAP Specification

☐ **Constructor**

1. *Map()*: create a new map that is undefined for all domain elements.

☐ **Checkers**

2. *isEmpty()*: return *true* if the map is empty (undefined for all domain elements), *false* otherwise.
3. *isDefined(d)*: return *true* if the image of *d* is defined, *false* otherwise.

☐ **Manipulators**

4. *assign(d,c)*: assign *c* as the image of *d*.
5. *image(d)*: return the image of *d* if it is defined, otherwise throw an exception.
6. *deassign(d)*: if the image of *d* is defined return the image and make it undefined, otherwise throw an exception.

4. List-based Representation

A map can be considered to be a list of pairs. Providing this list is *finite*, it can be implemented using one of the techniques used to implement List.

Better still, it can be built *using* List!

(Providing it can be done efficiently — recall example of *overwrite*, using *insert* and *delete*, in text editor based on List.)

Question: Which List ADT should we use?

- Require arbitrarily many assignments.
- Do we need *previous*?

Implementation. . .

```
public class MapLinked {

  private ListLinked list;

  public MapLinked () {
    list = new ListLinked();
```


4.1 Pairs

We said a (finite) map could be considered a list of pairs - need to define a Pair object. . .

```
public class Pair {

  public Object item1;       // the first item (or domain item)
  public Object item2;       // the second item (or codomain item)

  public Pair (Object i1, Object i2) {
    item1 = i1;
    item2 = i2;
    }
```

```java
  // determine whether this pair is the same as the object passed
  // assumes appropriate ‘‘equals’’ methods for the components
  public boolean equals(Object o) {
    if (o == null)  return false;
    else if (!(o instanceof Pair))  return false;
    else  return item1.equals( ((Pair)o).item1) &&
                 item2.equals( ((Pair)o).item2);
  }

  // generate a string representation of the pair
  public String toString() {
    return "< "+item1.toString()+" , "+item2.toString()+" >";
  }
}
```


4.2 Example — implementation of image

```java
public Object image (Object d) throws ItemNotFound {
  WindowLinked w = new WindowLinked();
  list.beforeFirst(w);
  list.next(w);
  while (!list.isAfterLast(w) &&
         !((Pair)list.examine(w)).item1.equals(d) )  list.next(w);
  if (!list.isAfterLast(w))  return ((Pair)list.examine(w)).item2;
  else throw new ItemNotFound("no image for object passed");
}
```

Notes:

1. `!list.isAfterLast(w)` must precede `list.examine(w)` in the condition for the loop — why??
2. Note use of parentheses around casting so that the field reference (eg `.item1`) applies to the cast object (`Pair` rather than `Object`).

3. Assumes appropriate *equals* methods for each of the items in a pair. Default is *equals* method inherited from `Object` — very conservative, assumes an object is only equal to itself. Many methods (eg `String`, `Character`, `Integer`,...) override this with their own. (We gave an example for `Pair`.)

4.3 Performance

Map and *isEmpty* make trivial calls to constant-time List commands.

The other 4 operations all require a sequential search within the list $\Rightarrow$ linear in the size of the defined domain ($O(n)$)

(Note — assumes constant-time List operations $\Rightarrow$ no use of *previous*.)

Performance using (singly linked) List ADT

Operation	
Map	1
isEmpty	1
isDefined	n
assign	n
image	n
deassign	n

If the maximum number of pairs is predefined, and we can specify a total ordering on the domain, better efficiency is possible. . .

5. Sorted-block Representation

Some of the above operations take linear time because they need to search for a domain element. The above program does a linear search.

Q: Are any more efficient searches available for arbitrary *linked* list?

5.1 Party Games. . .

Q: I've chosen a number between 1 and 1000. What is it?

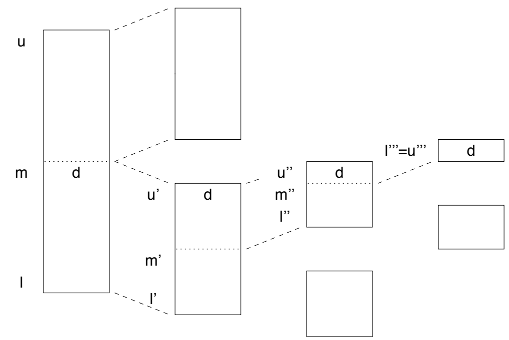
Q: I've chosen a number between 1 and 1000. If you make an incorrect guess I'll tell whether its higher or lower. You have 10 guesses. What is it?

Q: I'm going to choose a number between 1 and n. You have 5 guesses. What is the maximum value of n for which you are certain to get my number right?

Exercise: Write a recursive Java method *guessrange(m)* that returns the maximum number n for which you can always obtain a correct answer with m guesses.

5.2 Binary Search

An algorithm for binary search. . .



Assume `block` is defined as:

```
private Pair[] block;
```

Then binary search can be implemented as follows. . .


```
// recursive implementation of binary search
// uses String representations generated by toString()
// for comparison
// returns index to the object if found, or -1 if not found

protected int bSearch (Object d, int l, int u) {
  if (l == u) {
    if (d.toString().compareTo(block[l].item1.toString()) == 0)
      return l;
    else return -1;
  }
  else {
    int m = (l + u) / 2;
    if (d.toString().compareTo(block[m].item1.toString()) <= 0)
      return bSearch(d,l,m);
    else  return bSearch(d,m+1,u);
  }
}
```


Note: `compareTo` is an instance method of String — returns 0 if its argument matches the String, a value < 0 if the String is lexicographically less than the argument, and a value > 0 otherwise.

Exercise: Can `bSearch` be implemented using only the abstract operations of the List ADT?

5.3 Performance of Binary Search

We will illustrate performance in two ways.

One way of looking at the problem, to get a feel for it, is to consider the biggest list of pairs we can find a solution for with m calls to `bSearch`.

Calls to `bSearch`	Size of list
1	1
2	$1 + 1$
3	$2 + 1 + 1$
4	$4 + 2 + 1 + 1$
⋮	
m	$(2^{m-2} + 2^{m-3} + \cdots + 2^1 + 2^0) + 1$ $= (2^{m-1} - 1) + 1$ $= 2^{m-1}$

That is, $n = 2^{m-1}$ or $m = \log_2 n + 1$.

This view ignores the "intermediate" size lists — those which aren't a maximum size for a particular number of calls.

An alternative is to look at the number of calls needed for increasing input size. Can be expressed as a recurrence relation...

$$
\begin{array}{lcll}
T_1 & = & 1 & \\ \hline
T_2 & = & 1+T_1 & = 2 \\ \hline
T_3 & = & 1+T_2 & = 3 \\
T_4 & = & 1+T_2 & = 3 \\ \hline
T_5 & = & 1+T_3 & = 4 \\
T_6 & = & 1+T_3 & = 4 \\
T_7 & = & 1+T_4 & = 4 \\
T_8 & = & 1+T_4 & = 4 \\ \hline
T_9 & = & 1+T_5 & = 5 \\
 & & \vdots &
\end{array}
$$

The rows for which n is an integer power of 2...

$$
\begin{array}{lcll}
T_1 & = & 1 & \\ \hline
T_2 & = & 1+T_1 & = 2 \\ \hline
T_4 & = & 1+T_2 & = 3 \\ \hline
T_8 & = & 1+T_4 & = 4 \\ \hline
 & & \vdots &
\end{array}
$$

...correspond to those in the earlier table.

For these rows we have

$$
\begin{aligned}
T_{2^0} &= 1 \\
T_{2^m} &= 1 + T_{2^{m-1}} \\
&= 1 + 1 + T_{2^{m-2}} \\
&\ \vdots \\
&= \underbrace{1 + 1 + \cdots + 1}_{m+1 \texttt{ times}} \\
&= m + 1
\end{aligned}
$$

Substituting $n = 2^m$ or $m = \log_2 n$ once again gives

$$T_n = \log_2 n + 1.$$

What about the cases where n is not an integer power of 2?
$\Rightarrow$ Exercises.

It can be shown (see Exercises) that T_n is $O(\log n)$.

6. Comparative Performance of Operations

isDefined and *image* simply require binary search, therefore they are $O(\log n)$ — much better than singly linked list representation.

However, since the block is sorted, both *assign* and *deassign* may need to move blocks of items to maintain the order. Thus they are

$$\max(O(\log n), O(n)) = O(n).$$

In summary...

Operation	Linked List	Sorted Block
Map	1	1
isEmpty	1	1
isDefined	n	$\log n$
assign	n	n
image	n	$\log n$
deassign	n	n

Sorted block may be best choice if:

1. map has fixed maximum size
2. domain is totally ordered
3. map is fairly static — mostly reading (*isDefined*, *image*) rather than writing (*assign*, *deassign*)

Otherwise linked list representation will be better.

7. Summary

- A map (or function) is a many-to-one binary relation.
- Implementation using linked list
  - can be arbitrarily large
  - reading from and writing to the map takes linear time
- Sorted block implementation
  - fixed maximum size
  - requires ordered domain
  - reading is logarithmic, writing is linear

Data Structures and Algorithms

Topic 12

Arrays

- Arrays as a subtype of maps
- Array specification
- Lexicographically ordered representations
- Shell-ordered representation
- Extendibility
- Performance

Reading: Wood, Chapter 4

1. Arrays as Maps

We have seen two representations for maps

- linked list — linear time accesses
- sorted block — logarithmic for reading, linear for writing

One very frequently used subtype of the map is an array. An array is simply a map (function) whose domain is a cross product of (that is, tuples from) sets of ordinals — usually integers.

Unless stated otherwise we will assume all domain items are tuples of integers.

eg. The array

	1	**2**	**3**	**4**	**5**
false	6.6	2.8	0.4	6.0	0.1
true	3.4	7.2	9.6	4.0	9.9

could be represented by the map

$$\{\langle\langle 0,1\rangle, 6.6\rangle, \langle\langle 0,2\rangle, 2.8\rangle, \langle\langle 0,3\rangle, 0.4\rangle, \ldots,$$
$$\ldots, \langle\langle 1,4\rangle, 4.0\rangle, \langle\langle 1,5\rangle, 9.9\rangle\}$$

We will also assume the arrays are bounded in size, so we can store the items in a contiguous block of memory locations. (This can be simulated in Java using a 1-dimensional array.)

An *addressing function* can be used to translate the array indices into the actual location of the item.

Accesses are more efficient for this subtype of maps — *constant time in all operations*.

⇒ good example of a subtype over which operations are more efficient.

2. Specification

☐ **Constructors**

1. *Array()*: creates a new array that is undefined everywhere

☐ **Manipulators**

2. *assign(d,c)*: assigns c as the image of d
3. *image(d)*: returns the image of tuple *d* if it is defined, otherwise throws an exception

3. Lexicographically Ordered Representations

Lexicographic Ordering with 2 Indices

Pair $\langle i,j\rangle$ is *lexicographically earlier* than $\langle i',j'\rangle$ if $i < i'$ or ($i = i'$ and $j < j'$).

Best illustrated by an array with indices of type char:
first index: `a,...,d`
second index: `a,...,e`

Then entries are indexed in the order

$$\langle \mathtt{a},\mathtt{a}\rangle, \langle \mathtt{a},\mathtt{b}\rangle, \langle \mathtt{a},\mathtt{c}\rangle, \langle \mathtt{a},\mathtt{d}\rangle, \langle \mathtt{a},\mathtt{e}\rangle, \langle \mathtt{b},\mathtt{a}\rangle, \langle \mathtt{b},\mathtt{b}\rangle, \ldots \langle \mathtt{d},\mathtt{d}\rangle, \langle \mathtt{d},\mathtt{e}\rangle$$

⇒ ‘alphabetic’ order (*lexicon* ≈ dictionary)

	a	b	c	d	e
a	1	2	3	4	5
b	6	7	8	9	10
c	11	12	13	14	15
d	16	17	18	19	20

Also called *row-major* order.

Used in, eg, Fortran compilers

Implementation straightforward — indexed block (from 1 to 20 in the previous example).

Wish to access entries in constant time.

Addressing function $\alpha : 1..m \times 1..n \to \mathcal{N}$

$$\alpha(i,j) = (i-1) \times n + j \qquad 1 \le i \le m,\ 1 \le j \le n$$

Exercise

```
public class ArrayLexic {
  private Object[] block;
  private int numrows, numcolumns;

  public ArrayLexic(int m, int n) {
    block = new Object[m*n+1];   // start using array at 1 not 0
    numrows = m;
    numcolumns = n;
  }

  public void assign(PairInt indices, Object ob) throws OutOfBounds {
    if (1 <= indices.item1 && indices.item1 <= numrows &&
        1 <= indices.item2 && indices.item2 <= numcolumns)

    else throw new OutOfBounds("array indices out of bounds");
  }
```

Constant time?

Lexicographic Ordering with d Indices — Not Examinable

d-tuple $\langle i_1, \ldots, i_d \rangle$ is *lexicographically earlier* than $\langle i'_1, \ldots, i'_d \rangle$ if there is a k, $1 \le k \le d$, such that $i_1 = i'_1, i_2 = i'_2, \ldots, i_{k-1} = i'_{k-1}$ and $i_k < i'_k$.

eg. Assume $i_1, \ldots, i_d$ are all indexed over the range $['\mathsf{a}'..'\mathsf{c}']$

Index order...

$$\begin{array}{c}
\langle \mathsf{a}, \mathsf{a}, \ldots, \mathsf{a}, \mathsf{a} \rangle, \langle \mathsf{a}, \mathsf{a}, \ldots, \mathsf{a}, \mathsf{b} \rangle, \langle \mathsf{a}, \mathsf{a}, \ldots, \mathsf{a}, \mathsf{c} \rangle, \\
\langle \mathsf{a}, \mathsf{a}, \ldots, \mathsf{b}, \mathsf{a} \rangle, \langle \mathsf{a}, \mathsf{a}, \ldots, \mathsf{b}, \mathsf{b} \rangle, \langle \mathsf{a}, \mathsf{a}, \ldots, \mathsf{b}, \mathsf{c} \rangle, \\
\vdots \\
\langle \mathsf{c}, \mathsf{c}, \ldots, \mathsf{b}, \mathsf{a} \rangle, \langle \mathsf{c}, \mathsf{c}, \ldots, \mathsf{b}, \mathsf{b} \rangle, \langle \mathsf{c}, \mathsf{c}, \ldots, \mathsf{b}, \mathsf{c} \rangle, \\
\langle \mathsf{c}, \mathsf{c}, \ldots, \mathsf{c}, \mathsf{a} \rangle, \langle \mathsf{c}, \mathsf{c}, \ldots, \mathsf{c}, \mathsf{b} \rangle, \langle \mathsf{c}, \mathsf{c}, \ldots, \mathsf{c}, \mathsf{c} \rangle
\end{array}$$

Addressing function

$$\begin{aligned}\alpha(i_1, i_2, \ldots, i_d) &= (i_1 - 1) \times m_2 \times \cdots \times m_d \\ &+ (i_2 - 1) \times m_3 \times \cdots \times m_d \\ &\vdots \\ &+ (i_{d-1} - 1) \times m_d \\ &+ i_d\end{aligned}$$

for $1 \leq i_1 \leq m_1, \ 1 \leq i_2 \leq m_2, \ldots, 1 \leq i_d \leq m_d$

Complexity — Not Examinable

Indices can be computed in $2d$ arithmetic operations if some constant terms are precomputed.

Let $a_{k,d} = m_k \times m_{k+1} \times \cdots \times m_{d-1} \times m_d, \ 2 \leq k \leq d$, then the addressing function can be rewritten

$$\begin{aligned}\alpha(i_1, i_2, \ldots, i_d) &= (i_1 - 1).a_{2,d} + \cdots + (i_{d-1} - 1).a_{d,d} + i_d \\ &= i_1.a_{2,d} + \cdots + i_{d-1}.a_{d,d} + i_d.1 - (a_{2,d} + \cdots + a_{d,d}) \\ &= a_{1,d} + i_1.a_{2,d} + \cdots + i_{d-1}.a_{d,d} + i_d\end{aligned}$$

where $a_{1,d} = -a_{2,d} - a_{3,d} - \cdots - a_{d,d}$.

The constants $a_{k,d}, \ \ 1 \leq k \leq d$, can be precomputed when the array is created

$\Rightarrow$ to calculate any location index takes at most $d - 1$ multiplications and d additions!

Constant time for any fixed d.

Reverse-lexicographic Order

Similar to lexicographic, but indices swapped around...

Pair $\langle i, j \rangle$ is *reverse-lexicographically earlier* than $\langle i', j' \rangle$ if $j < j'$ or ($j = j'$ and $i < i'$).

	a	b	c	d	e
a	1	5	9	13	17
b	2	6	10	14	18
c	3	7	11	15	19
d	4	8	12	16	20

Also called *column-major* order.

Addressing function

$$\alpha(i, j) = (j - 1) \times m + i \qquad 1 \leq i \leq m, \ 1 \leq j \leq n$$

Used in, eg, Pascal compilers

4. Shell-ordered Representation

An alternative to lexicographic ordering — we will see later that it has advantages in terms of extendibility.

	a	b	c	d	e
a	1	2	5	10	17
b	4	3	6	11	18
c	9	8	7	12	19
d	16	15	14	13	20
	25	*24*	*23*	*22*	*21*

Built up shell by shell. kth shell contains indices $\langle i, j \rangle$ such that $k = \max(i, j)$.

Notice that the kth shell "surrounds" a block containing $(k-1)^2$ cells, and forms a block containing k^2 cells

$\Rightarrow$ To find entries in the first half of the shell, add to $(k-1)^2$. To find entries in the second half of the shell, subtract from k^2.

$$\alpha(i,j) = \begin{cases} (k-1)^2 + i & i < k \\ k^2 + 1 - j & \text{otherwise} \end{cases} \qquad k = \max(i,j).$$

Disadvantage

May waste a lot of space. . .

	a			
a	1	*2*	*5*	*10*
b	4	*3*	*6*	*11*
c	9	*8*	*7*	*12*
d	16	*15*	*14*	*13*

Worst case is a one-dimensional array of size n — wastes $n^2 - n$ cells.

A related problem occurs with all these representations when only a small number of the entries are used

eg. matrices in which most entries are zero

In this case more complex schemes can be used — trade space for performance. See Wood, Sec. 4.4.

Advantage

- *All* arrays use the same addressing function — independent of number of rows and columns.
- Extendibility. . .

5. Extendibility

In lexicographic ordering new rows can be added (if memory is available) *without changing* the values assigned to existing cells by the addressing function.

	a	b	c	d	e
a	1	2	3	4	5
b	6	7	8	9	10
c	11	12	13	14	15
d	16	17	18	19	20
e	21	22	23	24	25
f	26	27	28	29	30

$$\alpha(i,j) = (i-1) \times \underbrace{n}_{\text{no change}} + j \qquad 1 \le i \le m,\ 1 \le j \le n$$

We say the lexicographic addressing function is *row extendible*.

Adding a row takes $O(\text{size of row})$.

However it is not *column extendible*. Adding a new column means changing the values, *and hence locations*, of existing entries.

Q: What is an example of a worst case array for adding a column?

This is $O(\text{size of array})$ time operation.

Similarly, reverse lexicographic ordering is column extendible. . .

	a	b	c	d	e	f	g
a	1	5	9	13	17	21	25
b	2	6	10	14	18	22	26
c	3	7	11	15	19	23	27
d	4	8	12	16	20	24	28

$$\alpha(i,j) = (j-1) \times \underbrace{m}_{\text{no change}} + i \qquad 1 \le i \le m,\ 1 \le j \le n$$

. . . but not row extendible.

Shell ordering, on the other hand, is both row and column extendible. . .

	a	b	c	d	e	f
a	1	2	5	10	17	26
b	4	3	6	11	18	27
c	9	8	7	12	19	28
d	16	15	14	13	20	29
e	25	24	23	22	21	30
	36	*35*	*34*	*33*	*32*	*31*

This is because the addressing function is independent of m and n. . .

$$\alpha(i,j) = \begin{cases} (k-1)^2 + i & i < k \\ k^2 + 1 - j & \text{otherwise} \end{cases} \qquad k = \max(i,j).$$

for $1 \le i \le m,\ 1 \le j \le n$.

6. Performance Table

Operation	Lexicographic	Shell
Array	1	1
Assign	1	1
Image	1	1

7. Summary

Arrays are a commonly used subtype of maps which can be treated more efficiently.

- Can be implemented using a block and an addressing function.
- Choice of addressing functions — lexicographic, reverse-lexicographic, shell, etc
- Can be implemented efficiently — constant time in all operations.
- Shell addressing function is both row and column extendible, but may be an inefficient use of space.

Data Structures and Algorithms

Topic 13

Trees

- Why trees?
- Binary trees
  - definitions: size, height, levels, skinny, complete
- Trees, forests and orchards
- Tree traversal
  - depth-first, level-order
  - traversal analysis

Reading: Wood, Sections 5.1 to 5.4.

1. Why Study Trees?

Wood...

"Trees are ubiquitous."

Examples...

genealogical trees	organisational trees
biological hierarchy trees	evolutionary trees
population trees	book classification trees
probability trees	decision trees
induction trees	design trees
graph spanning trees	search trees
planning trees	encoding trees
compression trees	program dependency trees
expression/syntax trees	gum trees
⋮	⋮

Also, *many other data structures are based on trees!*

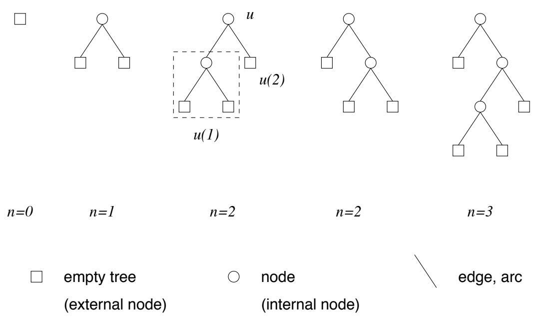
2. Binary Trees

Definitions

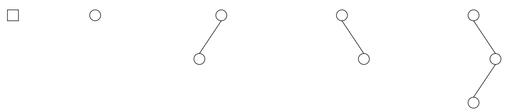
A *binary (indexed) tree* T of n nodes, $n \geq 0$, either:

- *is empty*, if $n = 0$, *or*
- *consists of a* root node u *and two binary trees* $u(1)$ and $u(2)$ of n_1 and n_2 nodes respectively such that $n = 1 + n_1 + n_2$.
  - $u(1)$: *first* or *left subtree*
  - $u(2)$: *second* or *right subtree*

  The function u is called the *index*.



We will often omit external nodes. . .



More terminology. . .

Definition

Let w_1, w_2 be the roots of the subtrees u_1, u_2 of u. Then:

- u is the *parent* of w_1 and w_2.
- w_1, w_2 are the (*left* and *right*) *children* of u. $u(i)$ is also called the i^{th} child.
- w_1 and w_2 are *siblings*.

Grandparent, *grandchild*, etc are defined as you would expect.

A *leaf* is an (internal) node whose left and right subtrees are both empty (external nodes).

The external nodes of a tree define its *frontier*.

In the following assume T is a tree with $n \geq 1$ nodes.

Definition

Node v is a *descendent* of node u in T if:

1. v is u, or
2. v is a child of some node w, where w is a descendent of u.

Proper descendent: $v \neq u$

Left descendent: u itself, or descendent of left child of u

Right descendent: u itself, or descendent of right child of u

Q: How would you define "v is *to the left of* u"?

Q: How would you define descendent without using recursion?

2.1 Size and Height of Binary Trees

The *size* of a binary tree is the number of (internal) nodes.

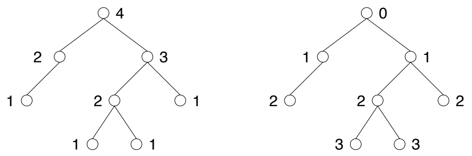
The *height* of a binary tree T is the length of the longest chain of descendents. That is:

- 0 if T is empty,
- $1 + \max(height(T_1), height(T_2))$ otherwise, where T_1 and T_2 are subtrees of the root.

The height of a node u is the height of the subtree rooted at u.

The *level* of a node is the "distance" from the root. That is:

- 0 for the root node,
- 1 plus the level of the node's parent, otherwise.

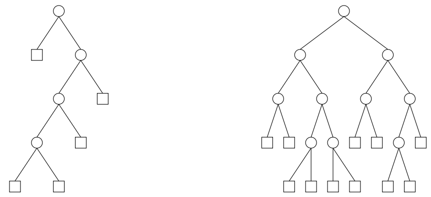


2.2 Skinny and Complete Trees

Since we will be doing performance analyses of tree representations, we will be interested in worst cases for height vs size.

skinny — every node has at most one child (internal) node

complete (fat) — external nodes (and hence leaves) appear on at most two adjacent levels

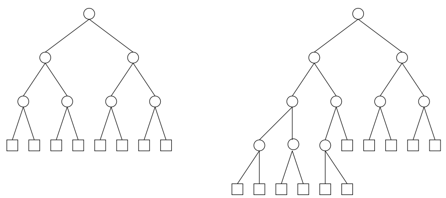


For a given size, skinny trees are the highest possible, and complete trees the lowest possible.

We also identify the following subclasses of complete:

perfect — all external nodes (and leaves) on one level

left-complete — leaves at lowest level are in leftmost position



2.3 Relationships between Height and Size

The above relationships can be formalised/extended to the following:

1. A binary tree of height h has size at least h.
2. A binary tree of height h has size at most $2^h - 1$.
3. A binary tree of size n has height at most n.
4. A binary tree of size n has height at least $\lceil \log(n+1) \rceil$.

Exercise

For each of the above, what class of binary tree represents an upper or lower bound? (For example, for (1), what sort of tree represents a lower bound on size for a given height?)

Exercise

Prove (2).

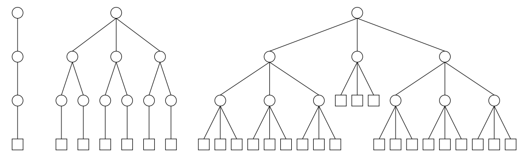
3. Trees, Forrests and Orchards

A general *tree* or *multiway (indexed) tree* is defined in a similar way to a binary tree except that a parent node does not need to have exactly two children.

Definition

A *multiway (indexed) tree* T of n nodes, $n \geq 0$, either:

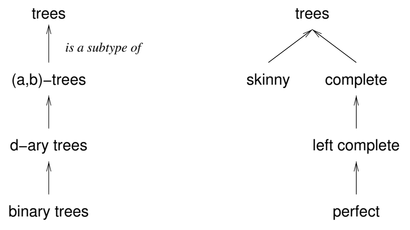
- is empty, if $n = 0$, or
- consists of a root node u, an integer $d \geq 1$ called the *degree* of u, and d multiway trees $u(1), u(2), \ldots, u(d)$ with sizes $n_1, n_2, \ldots, n_d$ respectively such that
$$n = 1 + n_1 + n_2 + \cdots + n_d.$$



A tree is a *d-ary tree* if $d_u = d$ for all (internal) nodes u. We have already looked at binary (2-ary) trees. Above is a unary (1-ary) tree and a ternary (3-ary) tree.

A tree is an (a,b)*-tree* if $a \leq d_u \leq b$, $(a,b \geq 1)$, for all u. Thus the above are all (1,3)-trees, and a binary tree is a (2,2)-tree.

Some trees of tree types!



3.1 Forests and Orchards

Removing the root of a tree leaves a collection of trees called a *forest*. An ordered forest is called an *orchard*. Thus:

forest — (possibly empty) set of trees

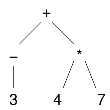
orchard — (possibly empty) queue or list of trees

3.2 Annotating Trees

The trees defined so far have no values associated with nodes. In practice it is normally such values that make them useful.

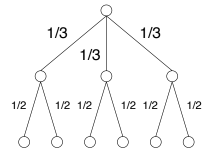
We call these values *annotations* or *labels*.

eg. a *syntax* or *formation* tree for the expression $-3 + 4 * 7$



eg. The following is a probability tree for a problem like:

> "Of the students entering a language course, one third study French, one third Indonesian and one third Warlpiri. In each stream, half the students choose project work and half choose work experience. What is the probability that Björk, a student on the course, is doing Warlpiri with work experience?"



In examples such as this one it often seems more natural to associate labels with the "arcs" joining nodes. However this is equivalent to moving the values down to the nodes.

As with List we will associate elements with the nodes.

4. Tree Traversals

Why traverse?

- search for a particular item
- test equality (isomorphism)
- copy
- create
- display

We'll consider two of the simplest and most common techniques:

depth-first — follow branches from root to leaves

breadth-first (level-order) — visit nodes level by level

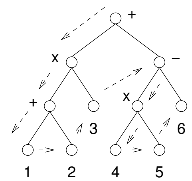
(More in Algorithms or Algorithms for AI...!)

4.1 Depth-first Traversal

Preorder Traversal

(Common garden "left to right", "backtracking", depth-first search!)

```
if(!t.isEmpty()) {
  visit root of t;
  perform preorder traversal of left subtree;
  perform preorder traversal of right subtree;
}
```



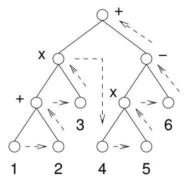
(Generates a *prefix expression*

$$+ \times + 1\,2\,3 - \times\, 4\,5\,6$$

Sometimes used because no brackets are needed — no ambiguity.)

Postorder Traversal

```
if(!t.isEmpty()) {
   perform postorder traversal of left subtree;
   perform postorder traversal of right subtree;
   visit root of t;
}
```



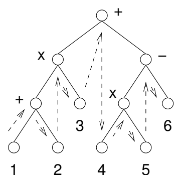
(Generates a *postfix expression*

$$1\,2 + 3 \times 4\;5 \times 6 - +$$

Also non-ambiguous — as used by, eg. HP calculators.)

Inorder Traversal

```
if(!t.isEmpty()) {
   perform inorder traversal of left subtree;
   visit root of t;
   perform inorder traversal of right subtree;
}
```



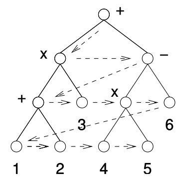
(Generates an *infix expression*

$$1 + 2 \times 3 + 4 \times 5 - 6$$

Common, easy to read, but ambiguous.)

4.2 Level-order (Breadth-first) Traversal

Starting at root, visit nodes level by level (left to right):



Doesn't suit recursive approach. Have to jump from subtree to subtree.

Solution:

- need to keep track of subtrees yet to be visited — data structure to hold (windows to) subtrees (or Orchard)
- each internal node visited spawns two new subtrees
- new subtrees visited *only after* those already waiting

$\Rightarrow$ Queue of (windows to) subtrees!

Algorithm

```
place tree (root window) on empty queue q;
while (!q.isEmpty()) {
  dequeue first item;
  if (!external node) {
    visit its root node;
    enqueue left subtree (root window);
    enqueue right subtree (root window);
  }
}
```


4.3 Traversal Analysis

Time

The traversals we have outlined all take $O(n)$ time for a binary tree of size n.

Since all n nodes must be visited, we require $\Omega(n)$ time
$\Rightarrow$ asymptotic performance cannot be improved.

Space

Depth-first: Recursive implementation requires memory (from Java's local variable stack) for each call $\Rightarrow$ proportional to height of tree

- worst case: skinny, size n implies height n
- expected case: much better (depends on distribution considered — see Wood Sec. 5.3.3)
- best case: *exercise...*

Iterative implementation is also possible.

Level-order: Require memory for queue.

Depends on tree *width* — maximum number of nodes on a single level.

Maximum length of queue is bounded by twice the width.

- best case: skinny, width 2
- worst case: *exercise...*

5. Summary

- Trees are not only common "in their own right" but form a basis for many other data structures.
- Definitions — binary trees, trees, forests, orchards, annotated trees
- Properties — size, height, level, skinny, complete, perfect, d-ary, (a, b)
- Covered important, common traversal strategies
  - depth-first: preorder, postorder, inorder
  - level-order (breadth-first)

Next — tree representations. . .

Data Structures and Algorithms

Topic 14

Tree Implementations

- Tree Specifications
- Block representation of Bintree
- Recursive representations of Bintree
- Recursive representation of Sbintree
- Representation of multiway Trees

Reading

Wood, Sections 5.5 to 5.9.

1. Specifications

Bintree

Just like List, we will have *windows* over nodes. The operations are similar, with *previous* and *next* replaced by *parent* and *child* and so on. Some are a little more complex because of the more complex structure. . .

□ **Constructor**

1. *Bintree()*: creates an empty binary tree.

□ **Checkers**

2. *isEmpty()*: returns *true* if the tree is empty, *false* otherwise.
3. *isRoot(w)*: returns *true* if *w* is over the root node (if there is one), *false* otherwise.
4. *isExternal(w)*: returns *true* if *w* is over an external node, *false* otherwise.
5. *isLeaf(w)*: returns *true* if *w* is over a leaf node, *false* otherwise.

□ **Manipulators**

6. *initialise(w)*: set *w* to the window position of the single external node if the tree is empty, or the window position of the root otherwise.
7. *insert(e,w)*: if w is over an external node replace it with an internal node with value e (and two external children) and leave w over the internal node, otherwise throw an exception.
8. *child(i,w)*: throw an exception if *w* is over an external node or *i* is not 1 or 2, otherwise move the window to the *i*-th child.
9. *parent(w)*: throw an exception if the tree is empty or w is over the root node, otherwise move the window to the parent node.
10. *examine(w)*: if w is over an internal node return the value at that node, otherwise throw an exception.
11. *replace(e,w)*: if w is over an internal node replace the value with e and return the old value, otherwise throw an exception.
12. *delete(w)*: throw an exception if *w* is over an external node or an internal node with no external children, otherwise replace the node under w with its internal child if it has one, or an external node if it doesn't.

Alternatives for *child*...

1. *left(w)*: throw an exception if *w* is over an external node, otherwise move the window to the left (first) child.
2. *right(w)*: throw an exception if *w* is over an external node, otherwise move the window to the right (second) child.

— can be convenient for binary trees, but does not extend to (multiway) trees.

Tree

Just modify to deal with more children (higher *branching*)...

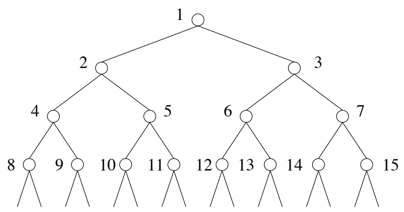
1. *degree(w)*: returns the degree of the node under w.
2. *child(i,w)*: throw an exception if *w* is over an external node or *i* is not in the range $1, \ldots, d$ where d is the degree of the node, otherwise move the window to the *i*-th child.

Orchard

Since an orchard is a list (or queue) of trees, an orchard can be specified simply using List (or Queue) and Tree (or Bintree)!

2. Block Representation of Bintree

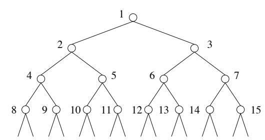
Based on an *infinite binary tree* — every internal node has two internal children...

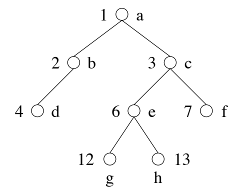


This is called a *level order* enumeration.
(Compare shell-ordered representation of an Array!)

Every binary tree is a *prefix* of the infinite binary tree — can be obtained by pruning subtrees.

Example...





1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	c	d		e	f					g	h		

Size of block needed is determined by height of tree.

Level-order representation is *implicit* — branches are not represented explicitly.

2.1 Time Performance

Level-order representation has the following properties:

1. $i(u) = 1$ iff u is the root.
2. Left child of u has index $2i(u)$.
3. Right child of u has index $2i(u) + 1$.
4. If u is not the root, then the parent of u has index $i(u)/2$ (where / is integer division).

These properties are important — allow constant time movement between nodes

$\Rightarrow$ all Bintree operations are constant time!

2.2 Space

Level-order representation can waste a great deal of space.

Q: What is the worst case for memory consumption?

Q: What is the best case for memory consumption?

A binary tree of size n may require a block of size $2^n - 1$

$\Rightarrow$ exponential increase in size!

3. Recursive Representations of Bintree

Basic Structure

Recall List:

- recursive definition
- recursive singly linked structure — one item, one successor

We can do the same with binary trees — difference is we now need *two* "successors".

Recall the (recursive) definition of a binary tree — can be briefly paraphrased as:

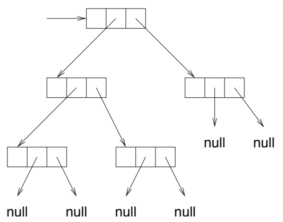
A binary tree either:

- is empty, or
- consists of a root node u and two binary trees $u(1)$ and $u(2)$. The function u is called the *index*.

It can be implemented as follows.

First, instead of "`Link`" use a `TreeCell`...

```java
public class TreeCell {

  public Object nodeValue;
  public TreeCell[] children;

  public TreeCell(Object v, TreeCell tree1, TreeCell tree2) {
    nodeValue = v;
    children = new TreeCell[2];
    children[0] = tree1;
    children[1] = tree2;
  }
}
```



The `children` array performs the role of the *index* u — it holds the "successors".

An alternative for binary trees is...

```java
public class TreeCell {

  public Object nodeValue;
  public TreeCell left;
  public TreeCell right;

  public TreeCell(Object v, TreeCell tree1, TreeCell tree2) {
    nodeValue = v;
    left = tree1;
    right = tree2;
  }
}
```

but this doesn't extend well to trees in general. The previous version can easily be extended to multiway trees by initialising larger arrays of children.

Windows

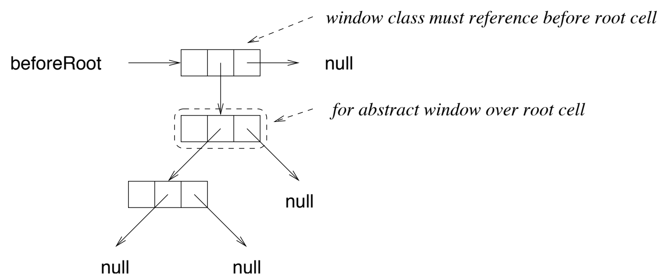
Just like Lists, we wish to allow multiple windows for manipulating Trees. We will therefore define a "companion" window class.

In the Notes and Exercise Sheets on Lists we considered a representation in which the window contained a member variable that referenced the cell previous to the (abstract) window position. This was so that *insertBefore* and *delete* could be implemented in constant time without moving data around.

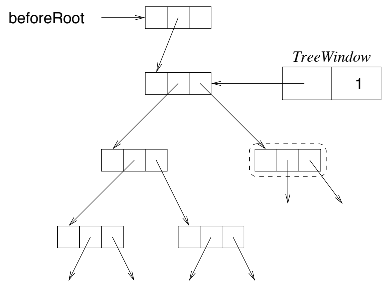
Similar problems arise in trees with *delete*, where we want to point the parent node to a different child.

We will use the same technique — the window class will store a reference to the *parent* of the (abstract) window node

$\Rightarrow$ requires a "before root" cell.



Since the parent has two children, we need to know which the window is over, so we include a branch number. . .




```
public class TreeWindow {
  public TreeCell parentnode;
  public int childnum;

  public TreeWindow () {}
}
```

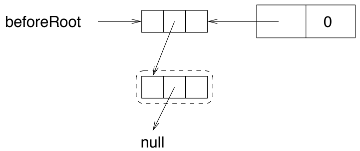
For example. . .

```
public void initialise(TreeWindow w) {
  w.parentnode = beforeRoot;
  w.childnum = 0;
}
```

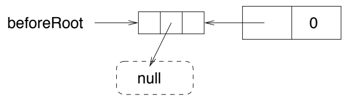

External nodes

Two choices:

1. If values are attached to external nodes, the external nodes must be represented by cells. They can be distinguished from internal nodes by a null reference as the left child.



2. If external nodes have no values they can be represented simply by null references. . .



We will assume external nodes do *not* store values, and represent them by null references.

3.1 Examples

Constructor

```
public BintreeLinked () {
  beforeRoot = new TreeCell(null, null, null);
}
```

Checkers

```
public boolean isEmpty() {return beforeRoot.children[0] == null;}
```

```
public boolean isExternal(TreeWindow w) {
  return w.parentnode.children[w.childnum] == null;
}
```



```
public boolean isLeaf(TreeWindow w) {
  return !isExternal(w)
    && w.parentnode.children[w.childnum].children[0] == null
    && w.parentnode.children[w.childnum].children[1] == null;
}
```


Manipulators

Exercises. . .

```
public Object examine(TreeWindow w) throws OutOfBounds {
  if (!isExternal(w))

  else throw new OutOfBounds("examining external node");
}
```

```
public void insert(Object e, TreeWindow w) throws OutOfBounds {
  if (isExternal(w))

  else
    throw new OutOfBounds("inserting over internal node");
}
```


3.2 Performance

Clearly all operations except *parent* can be implemented to run in constant time.

parent in Bintree is like *previous* in List.

Can be achieved in a similar manner to link coupling — search the tree from the before-root node. Recall traversals from Section 13!

Takes $O(n)$ time in worst case for binary tree of size n.

Q: What representation could we use to obtain a constant time implementation of *parent*?

3.3 Sbintree

Just like Simplist, if a tree only requires one window, we can implement it using reference reversal!

Analogous to Simplist (tho a bit more involved):

- implicit window
- constant time implementation of *parent*
- *initialise* is linear time, but constant time in the amortized case
- avoid stack memory for recursion during depth-first traversal

See Wood, Sec. 5.5.4.

4. Trees

Recursive representation can be extended to multiway trees — just increase the size of the `children` array...

```
public class TreeCell {
  public Object nodeValue;
  public TreeCell[] children;

  public TreeCell(int degree, Object v, TreeCell tree1,...) {
    nodeValue = v;
    children = new TreeCell[degree];
    children[0] = tree1;
    children[1] = tree2;
    .
    .
  }

}
```


5. Summary

- block representation of Bintree
  - time efficient — constant time in all ops
  - not space efficient — may waste nearly 2^n cells
- recursive representation of Bintree
  - a generalisation of List
  - choices for window and external node representations
  - *parent* is linear time (traversal), all other ops are constant time
- Sbintree
  - analogous to Simplist
  - implicit window, pointer reversal
  - *parent* constant time, *initialise* constant in amortized case
- Tree
  - generalisation of Bintree

Data Structures and Algorithms

Topic 15

Sets, Tables and Dictionaries

- What do we mean by sets, tables, and dictionaries?
- Set specification
- Set representations
  - characteristic function
  - lists
  - ordered lists
- Table specification
- Table representations

- Dictionary specifications
- Dictionary representations
  - Set-based representations
  - binary search trees

Reading

Wood, Chapter 8

1. Introduction

In this section we examine three ADTs: *sets*, *tables* and *dictionaries*, used to store collections of elements with no repetitions.

Note that these names are used (eg in different texts) for a range of similar ADTs — we define them as follows:

Set

- used when set-theoretic operations are required
- elements may or may not be ordered
- "typical" operations *isEmpty*, *insert*, *delete*, *isMember*
- "set-theoretic" operations *union*, *intersection*, *difference*, *size*, *complement*

Table

- simpler version of set without the set-theoretic operations
- elements assumed to be unordered

Dictionary

- like Table but assumes elements are totally ordered
- "order related" operations *isPredecessor*, *isSuccessor*, *predecessor*, *successor*, *range*

1.1 Elements, Records and Keys

Elements may be a single items, or "records" with unique *keys* (such as those typically found in databases).

We will usually talk about elements as if they are single items.

eg. "if $e_1 < e_2$ then..."

In the case of record elements this can be considered shorthand for

"if $k_1 < k_2$, where k_1 is the key of record e_1 and k_2 is the key of record e_2, then..."

1.2 Examples of Use

The following are examples of the (many) sorts of situations where the ADTs might be used:

Set

"I have one set of students who do CS223 and one set of students who do CS226. What is the set of students who do both?"

Table

"I begin with the set of students originally enrolled in CS223. These two students joined. This one withdrew. Is a particular student currently enrolled?"

Dictionary

"Here is the set of students enrolled in CS223 ordered by (exact) age. Which are the students between the ages of 18 and 20?"

2. Set Specification

☐ **Constructors**

1. *Set()*: create an empty set.

☐ **Checkers**

2. *isEmpty()*: returns *true* if the set is empty, *false* otherwise.
3. *isMember(e)*: returns *true* if *e* is a member of the set, *false* otherwise.

☐ **Manipulators**

4. *size()*: returns the cardinality of (number of elements in) the set.
5. *complement()*: returns the complement of the set (only defined for finite universes).
6. *insert(e)*: forms the union of the set with the singleton $\{e\}$
7. *delete(e)*: removes e from the set
8. *union(t)*: returns the union of the set with t.
9. *intersection(t)*: returns the intersection of the set with *t*.
10. *difference(t)*: returns the set obtained by removing any items that appear in t.
11. *enumerate()*: returns the "next" element of the set. Successive calls to *enumerate* should return successive elements until the set is exhausted.

3. Set Representations

Characteristic Function Representation

Assume A is a set from some universe U.

The *characteristic function* of A is defined by:

$$f(e) = \begin{cases} \mathit{true} \ \ (\text{or } \ 1) & e \in A \\ \mathit{false} \ \ (\text{or } \ 0) & \text{otherwise} \end{cases}$$

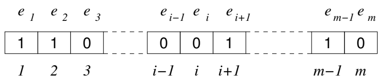
$\Rightarrow$ thus a set can be viewed as a boolean function.

If U is finite and '$\leq$' is a total order on U, the elements of U can be enumerated as the sequence

$$e_1, \ldots, e_m$$

where $e_i \leq e_j$ if $i < j$, and m is the cardinality of A.

The characteristic function maps this sequence to a sequence of 1s and 0s. Thus the set can be represented as a block of 1s and 0s, or a *bit vector*...



Sometimes called a *bitset* — eg. `java.util.BitSet`

Advantage

Translates set operations into efficient bit operations:

- *insert* — *or* the appropriate bit with 1
- *delete* — *and* the appropriate bit with 0
- *isMember* — is the (boolean) value of the appropriate bit
- *complement* — complement of a bit vector
- *union* — *or* two bit vectors
- *intersection* — *and* two bit vectors
- *difference* — *complement* and *intersection*

Also *enumerate* — can cycle through the m positions reporting 1s.

Performance

- *insert*, *delete*, *isMember* — constant providing index can be calculated in constant time
- *complement*, *union*, *intersection*, *difference* — $O(m)$; linear in size of *universe*
- *enumerate* — $O(m)$ for n calls, where n is size of set

  $\Rightarrow$ $O(\frac{m}{n})$ amortized over n calls

Disadvantages

- If the universe is large compared to the size of sets then:
  - the latter ops are expensive
  - large amount of space wasted
- Requires the universe to be bounded, totally ordered, and known in advance.

List Representation

An alternative is to represent the set as a list using one of the List representations. Here we assume there is not a total ordering on the elements.

Performance

Assume we have a set of size m.

insert, *delete*, *isMember* take $O(m)$ time — best that can be achieved in an unordered list (recall `eSearch`)

union — for each item in the first set, check if it is a member of the second, and if not, add it (to the result)

$\Rightarrow \quad O(mn)$ where m and n are the sizes of the two sets

Other set operations (*intersection*, *difference*) behave similarly.

Note that if both sets grow at the same rate (the worst case) the time performance is $O(n^2)$.

Inefficient because one list must be traversed for each element in the other. Can we traverse both at the same time...?

Ordered List Representation

If the universe is totally ordered, we can obtain more efficient implementations by merging the two in sorted order.

Assume A can be enumerated as $a_1, a_2, \ldots, a_m$ and B can be enumerated as $b_1, b_2, \ldots, b_n$.

Eg. *union*

```
i = 1; j = 1;
do {
   if (a_i == b_j)  add a_i to C and increment i and j;
   else add smaller of a_i and b_j to C and increment its index;
}
while (i <= m && j <= n);
add any remaining a_i's or b_j's to C
```


Exercise

Give pseudo-code for ops *intersection* and *difference*.

Performance

Each list is traversed once $\Rightarrow \quad O(m+n)$ time.

This is much better than $O(mn)$.

If m and n grow at the same rate (worst case) the time performance is now $O(n)$.

Note also that *isMember* is now $O(\log m)$ (recall `bSearch`)

4. Table Specification

The Table operations are a subset of the Set operations:

□ **Constructors**

1. *Table()*: create an empty table.

□ **Checkers**

2. *isEmpty()*: returns *true* if the table is empty, *false* otherwise.
3. *isMember(e)*: returns *true* if *e* is in the table, *false* otherwise.

□ **Manipulators**

4. *insert(e)*: forms the union of the table with the singleton $\{e\}$
5. *delete(e)*: removes e from the table

5. Table Representations

Since the Table operations are a subset of those of Set, the (unordered) List representations can be used.

insert, *delete*, *isMember* therefore take $O(m)$ time.

The more efficient List representations and the characteristic function representation are not available since the elements are assumed to be unordered.

The operations can be made more efficient by considering the probability distribution for accesses over the list and moving more probable (or more frequently accessed) items to the front — see Wood, Section 8.3.

Later we'll look in detail at a more efficient representation of tables using *hashing* in which such operations are close to constant time.

6. Dictionary Specification

□ **Constructors**

1. *Dictionary()*: creates an empty dictionary.

□ **Checkers**

2. *isEmpty()*: returns *true* if the dictionary is empty, *false* otherwise.
3. *isMember(e)*: returns *true* if *e* is a member of the dictionary, *false* otherwise.
4. *isPredecessor(e)*: returns *true* if there is an element in the dictionary that precedes *e* in the partial order, *false* otherwise.
5. *isSuccessor(e)*: returns *true* if there is an element in the dictionary that succeeds *e* in the partial order, *false* otherwise.

□ **Manipulators**

6. *insert(e)*: adds e (if not already present) to the dictionary in the appropriate position.
7. *predecessor(e,p)*: returns the largest element *p* that is smaller than *e*, if one exists, otherwise throws an exception.
8. *successor(e,s)*: returns the smallest element *s* that is larger than *e*, if one exists, otherwise throws an exception.
9. *range(p,s)*: returns the dictionary of all elements that lie between *p* and *s* (including *p* and *s* if present) in the ordering.
10. *delete(e)*: removes item e from the dictionary (if it exists).

7. Dictionary Representations

Representations based on Set

We have already seen two representations that can be used for Sets when there is a total ordering on the universe...

- characteristic function (bit vector) representation
  - time efficiency (eg $O(1)$ for *isMember*) gained by indexing directly to appropriate bits
  - bounded — universe fixed in advance
  - space wasted if universe is large compared with commonly occurring sets

- List based (ordered block) representation
  - time efficiency (eg $O(\log n)$ for *isMember*) comes from binary search
  - bounded
  - space usage may be poor if large block is set aside

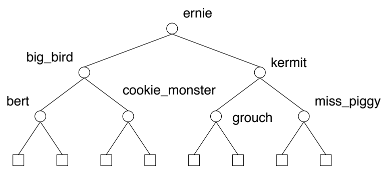
We now examine a representation which supports a binary-like search but is unbounded...

7.1 Binary Search Trees

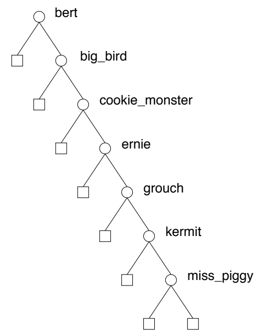
A *binary search tree* is a binary tree whose internal nodes are labelled with elements (or their keys) such that they satisfy the *binary search tree condition*:

> For every internal node u, all nodes in u's left subtree precede u in the ordering and all nodes in u's right subtree succeed u in the ordering.

eg.



eg.



7.2 Searching

If information is stored in a binary search tree a simple recursive "divide and conquer" algorithm can be used to find elements:

`if (t.isEmpty())` terminate unsuccessfully;
`else` {
  r becomes the element on the root node of `t`;
  `if` $(e$ equals $r)$ terminate successfully;
  `else if` $(e < r)$ repeat search on left subtree;
    `else` repeat search on right subtree;
}

7.3 Performance

Depends on the shape of the tree...

Exercise

- Best case is a perfect binary tree. What is the performance of *isMember*?
- Worst case is a skinny binary tree. What is the performance of *isMember*?

7.4 insert and delete

insert is fairly straightforward

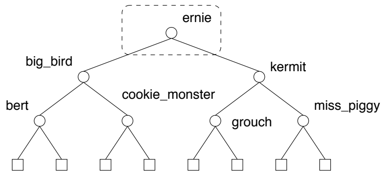
- perform a search for the element as above
- if the element is found take no further action
- if an empty node is reached insert a new node containing the element

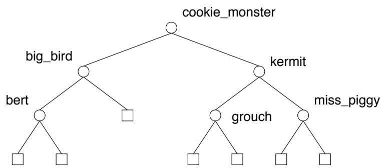
delete is straightforward if the element is found on a node with at least one external child — just use the standard Bintree *delete* operation

Otherwise:

1. replace the deleted element with its predecessor — note that the predecessor will always have an empty right child
2. delete the predecessor

eg. ... $\longrightarrow$





Note that the *delete* procedure described here has a tendency over time to skew the tree to the right — as we have seen this will make it less efficient.

Alternative — alternate between replacing with predecessor and successor.

In general, it is beneficial to try to keep the tree as "balanced" or "complete" as possible, to maintain search efficiency.

8. Summary

We have outlined 3 ADTs for use with collections of unique elements or records:

- Set — includes set-theoretic operations, elements may or may not be ordered
- Table — restriction of Set with fewer operations, elements assumed not ordered
- Dictionary — extension of Table, assumes ordering and contains "order-related" operations

We have covered a number of representations:

- List — can be used for unordered sets and tables
- ordered list (block) — can improve efficiency for ordered sets and can be used for bounded dictionaries
- characteristic function — can be very efficient for ordered sets and dictionaries in certain cases, bounded
- binary search tree — unbounded representation for dictionaries, efficiency better than List but depends on tree shape

Next — efficient representations for Tables...

Data Structures and Algorithms

Topic 16

Hash Tables

- Introduction to hashing — basic ideas
- Hash functions
  - properties, 2-universal functions, hashing non-integers
- Collision resolution
  - bucketing and separate chaining
  - open addressing
  - dynamic tables — linear hashing

Reading: Wood, Chapter 9

1. Introduction

The Table is one of the most commonly used data structures — central to databases and related information systems.

In the previous section we briefly examined a List representation for tables, but this had linear time access.

Can we do better?

We have seen a number of situations where constant time access to data can be achieved by indexing directly into a block. . .

eg. Array uses an addressing function

$$\alpha(i,j) = (i-1) \times n + j \qquad 1 \le i \le m,\ 1 \le j \le n$$

	a	b	c	d	e
a	1	2	3	4	5
b	6	7	8	9	10
c	11	12	13	14	15
d	16	17	18	19	20

eg. Set uses a characteristic function. . .

$$f(e) = \begin{cases} \textit{true} \ \ (\text{or } 1) & e \in A \\ \textit{false} \ \ (\text{or } 0) & \text{otherwise} \end{cases}$$

e_1	e_2	e_3	…	e_{i-1}	e_i	e_{i+1}	…	e_{m-1}	e_m
1	1	0		0	0	1		1	0
1	2	3		$i-1$	i	$i+1$		$m-1$	m

These approaches:

- often sacrifice space for time — space wasted by all the "holes"
- rely on the ordering of elements, which translates to an ordering of the memory block.

Can we improve on these?

- more compact use of space
- applicable to unordered information (eg Table)

$\Rightarrow$ *hashing. . .*

2. Basic Hashing

The direct indexing approaches above work by:

- setting aside a big enough block for all possible data items
- spacing these so that the address of any item can be found by a simple calculation from its ordinality

What if we use a block which is not big enough for all possible items?

- Addressing function must map all items into this space.
- Some items may get mapped to the same position $\Rightarrow$ called a *collision*.

Example

Suppose we fill out our Lotto coupons as follows. Each time we notice a positive integer in our travels, we calculate its remainder modulo 45 and add that to our coupon...

The first thing we see is a pizza brouchure containing the numbers 165, 93898500, 2, 13, 1690. These map to positions 30, 15, 2, 13, 25.

This fills 5 positions in our data store...

0	1	~~2~~	3	4	5	6	7	8
9	10	11	12	~~13~~	14	~~15~~	16	17
18	19	20	21	22	23	24	~~25~~	26
27	28	29	~~30~~	31	32	33	34	35
36	37	38	39	40	41	42	43	44

Next we ring up to order our Greek Vegetarian, and we're told it'll be ready in 15!

$\Rightarrow$ *collision*

We need a method for dealing with this. However...

Advantages:

- Once we allow collisions we have much more freedom in choosing an addressing function.
- It no longer matters whether we know an ordering over the items.

Exercise

Obtain an address from your name into the block 1...10 as follows:

- Count up the number of letters in your name.
- Add 1.
- Double it.
- Add 1.
- Double it.
- Subtract the number of letters in your name.
- Add the digits in your current number together.
- Square it.
- Add the digits in your number together.

What address did you get?

Such a function is called a *hash function*. It takes the item to be looked up or stored and "hashes it" into an address in the block, or *hash table*.

We will consider hash functions in more detail, and then consider methods for dealing with collisions.

3. Hash Functions

To begin with we'll assume that the element (or key) to be hashed is an integer. We need a function

$$h : \mathcal{N} \to 0 \ldots m-1$$

that maps it into a hash table `block` of size m.

Thus to store a table t of elements we would set

$$\texttt{block}[h(a)] = a$$

for all elements a in t, and fill the other elements of `block` with null references.

We call $h(a)$ the *home address* of a.

3.1 Properties of Hash Functions

A hash function that maps each item to a unique position is called *perfect*.

(Note that if we had an infinitely large storage block we could always design a perfect hash function.)

Since our hash functions will generally not be perfect, we want a function that distributes evenly over the hash table — that is, one that is not *biased*.

Q: What is an example of a "worst case" hash function in terms of bias?

3.2 2-universal Functions

In the Lotto example earlier we used the hash function

$$h(i) = \quad i \bmod 45.$$

A commonly used class of hash functions, called *2-universal* functions, extends this idea. . .

A *2-universal* hash function has the form

$$h(i) = ((c_1 i + c_2) \bmod p) \bmod m$$

where m is the size of the hash table, $p > m$ is a large prime ($p > 2^{20}$), $c_1 < p$ is a positive integer, and $c_2 < p$ is a non-negative integer.

— $(c_1 i + c_2) \bmod p$ "scrambles" i

— $\bmod m$ maps into the block

Small changes (eg to c_1) lead to completely different hash functions.

[Another 2-universal hash function that can be used in languages with bit-string operations...

Assume items are b-bit strings for some $b > 0$, and $m = 2^l$ for some $l > 0$. The *multiplicative hash function* has the form:

$$h(i) = (a\, i \bmod 2^b) \operatorname{div} 2^{b-1}$$

for odd a such that $0 < a \leq 2^b - 1$.

Advantage — **mod** and **div** operations can be evaluated by shifting rather than integer division $\Rightarrow$ very quick]

3.3 Hashing Non-integers

Non-integers are generally mapped (hashed!) to integers before applying the hash functions mentioned earlier.

Example

Assume we have a program with 3 variables

```
float abc, abd, bad;
```

and we wish to hash to a location to store their values. We could obtain an integer from:

- the length of each word (as in the earlier example) — will lead to a lot of collisions $\Rightarrow$ in this case all variables will hash to the same location
- the ordinality of the first character of each word — fewer collisions, but still poor $\Rightarrow$ the first two will hash to the same location

- summing the ordinality of all characters — likely to be even fewer collisions $\Rightarrow$ but here the last two will collide
- "weight" the characters differently, eg 3 times the first plus two times the second plus one times the third — collisions will be much rarer (but may still occur) $\Rightarrow$ no collisions in this set

Extending the weighting idea, a typical hash function for strings is to treat the characters as digits of an integer to some base b.

Assume we have a character string $s_1 s_2 \ldots s_k$. Then we calculate

$$[Ord(s_1).b^{k-1} + Ord(s_2).b^{k-2} + \ldots + Ord(s_k)b^0] \bmod 2^B$$

Here

- b is a small odd number, such as 37...
- $\bmod 2^B$ gives the least significant B bits of the result — eg 16 or 32.

Q: Why the *least* significant bits?

4. Collision Resolution Techniques

There are many variations on collision resolution techniques. We consider examples of three common types.

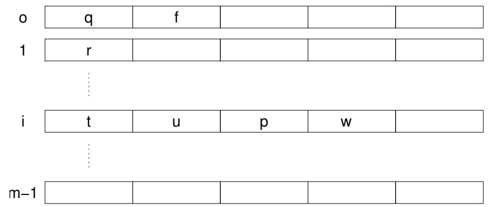
4.1 Bucketing and Separate Chaining

The simplest solution to the collision problem is to allow more than one item to be stored at each position in the hash table

⇒ associate a List with each hash table cell...

Bucketing

— each list is represented by a (fixed size) block.



"Advantage"

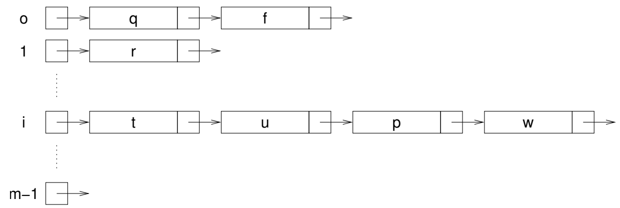
- Simple to implement — hash to address then search list.

Disadvantages

- Searching the List slows down Table access.
- Fixed size ⇒ may waste a lot of space (both in hash table and buckets).
- Buckets may *overflow!* ⇒ back where we started (a collision is just an overflow with a bucket size of 1).

Separate Chaining (variable size bucketing)

— each List is represented by linked list or *chain*.



Advantages

- Simple to implement.
- No overflow.

Disadvantages

- Searching the List slows down Table access.
- Extra space for pointers (if we are storing records of information the space used by pointers will generally be small compared to the total space used).
- Performance deteriorates as chain lengths increase.

[Performance

Worst case for separate chaining $\Rightarrow$ all items stored in a single chain. Worst case performance same as List: $O(n)$ — nothing gained!

But *expected case* performance is much better...

The *load factor* λ of a hash table is the number of items in the table divided by the size m of the table.

> *Assume that each entry in a hash table is equally likely to be accessed, and that each sequence of n insertions is equally likely to occur. Then a hash table that uses separate chaining and has load factor λ has the following expected case performance:*
>
> - $s(\lambda) = 2 + \lambda/2$ *probes (read accesses) for successful search*
> - $u(\lambda) = 2 + \lambda$ *probes for unsuccessful search*

Wood, Section 9.2]

4.2 Open Addressing

Separate chaining (and bucketing) require additional space. Yet there will normally be space in the table that is wasted.

Alternative $\Rightarrow$ **open addressing** methods

- store all items in the hash table
- deal with collisions by incrementing hash table index, with wrap-around

Linear probing — increment hash index by one (with wrap-around) until the item, or *null*, is found.

Problem — items tend to "cluster".

Double hashing — increment hash index using an "increment hash function"! $\Rightarrow$ may jump to anywhere in table.

Advantages

- All space in the hash table can be used.

Disadvantages

- Insertions limited by size of table.
- Deletions are problematic...

Deleting items means others may not be able to be reached — requires reorganizing table, or marking (flagging) items as deleted.

The latter is most common, but means erosion of space in the hash table.

4.3 Dynamic Tables — Linear Hashing

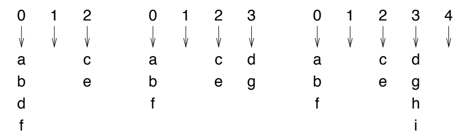
Finally, there are methods that consider the hash table to be dynamic rather than static!

Linear hashing is an extension of separate chaining — rather than allowing the variable-length buckets (chains) to grow indefinitely, we limit the average size of the buckets.

- Insertions: if average chain size exceeds a predefined upper bound, split the "next" unsplit bucket, and hash the items in the bucket by a function with double the previous base (ie m, $2m$, $4m$,...).
- Deletions: if average bucket size drops below a predefined minimum, and the table is no smaller than the original table, shrink the table.

Example

Assume table is initially of size 3, and maximum loading is 2...



Disadvantages

- (More complicated to code.)
- Requires movement of items.

Advantages

- Maximum load is maintained — improves expected efficiency.
- Insertions — no overflow, not bounded by size of table.
- Deletions — no erosion.

This approach is used by `java.util.Hashtable` — have a look at its API documentation.

5. Summary

Hash tables can be used to:

- improve the space requirements of some ADTs for which bounded representations are suitable
- improve the time efficiency of some ADTs, such as Table, which require unbounded representations

We have seen a number of methods for collision resolution in hash tables:

- bucketing and separate chaining
- open addressing, including linear probing and double hashing
- dynamic methods, such as linear hashing

5. Summary

Note that while performance can be very good, *this is not a panacea!* For many applications, such as those naturally represented by trees, hashing would lose the structure.