CITS2200 Data Structures and Algorithms

Topic 11

Tree Implementations

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

Reading: Weiss, Chapter 18

1. Specifications

Binary Tree (Bintree)

Just like the list ADT, 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.

Note: as with the list ADT, the `Window` class can be replaced by a `treeIterator` to navigate and manipulate the tree.

Tree

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

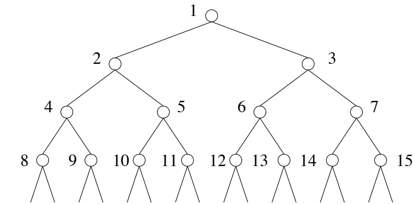
1. *degree(w)*: returns the degree of the node under w.
2. *child(i,w)*: throw an exception if *w* is over an external node or *i* is not in the range $1, \ldots, d$ where d is the degree of the node, otherwise move the window to the *i*-th child.

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

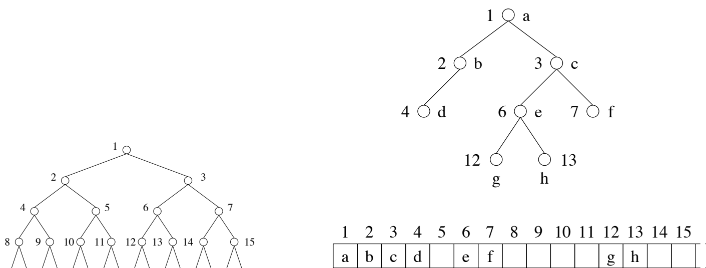
Based on an *infinite binary tree* — every internal node has two internal children...



This is called a *level order* enumeration.

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

Example...



The size of block needed is determined by the 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*.

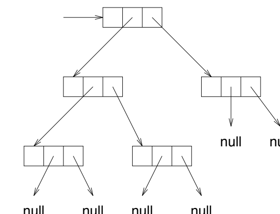
It can be implemented as follows.

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


```
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. . .

```
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

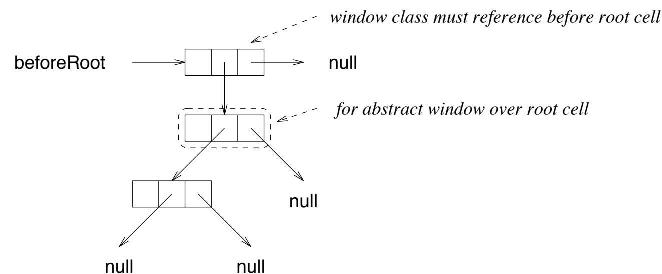
Just like lists, we wish to allow multiple windows for manipulating trees. We will therefore define a "companion" window class.

In the notes and exercises 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.

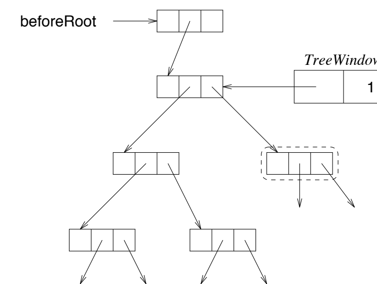
Similar problems arise in trees with *delete*, where we want to point the parent node to a different child.

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 () {}
}
```

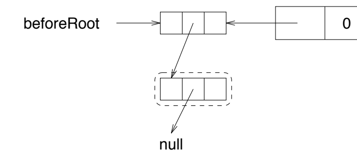
For example. . .

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

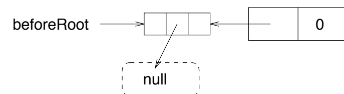

External Nodes

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 Topic 10!

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 the simplist ADT, if a tree only requires one window, we can implement it using reference reversal!

Analogous to Simplist (although a bit more involved):

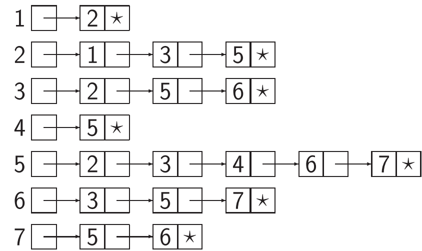
- implicit window
- constant time implementation of *parent*
- *initialise* is linear time, but constant time in the amortized case
- avoid stack memory for recursion during depth-first traversal

Representation of graphs

Two ways to represent a graph — adjacency lists or an adjacency matrix.

Adjacency lists The graph G is represented by an array of $|V(G)|$ linked lists, with each list containing the neighbours of a vertex.

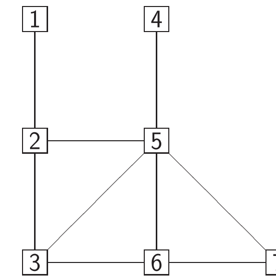
Therefore we would represent G_4 as follows:



This Requires two list elements for each edge and thus the space required is $O(|V(G)| + |E(G)|)$.

For comparison...

...the graph G_4.



Note: In general to avoid writing $|V(G)|$ and $|E(G)|$ we shall simply put $V = |V(G)|$ and $E = |E(G)|$.

Adjacency matrix

The *adjacency matrix* of a graph G is a $V \times V$ matrix A where the rows and columns are indexed by the vertices and such that $A_{ij} = 1$ if and only if vertex i is adjacent to vertex j.

For graph G_4 we have the following

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

The adjacency matrix representation uses $O(V^2)$ space. For a *sparse* graph E is much less than V^2, and hence we would normally prefer the adjacency list representation while for a *dense* graph E is close to V^2 and the adjacency matrix representation is preferred.

More on the two representations

For small graphs or those without weighted edges it is often better to use the adjacency matrix representation anyway.

It is also easy and more intuitive to define adjacency matrix representations for directed and weighted graphs.

However your final choice of representation depends precisely what questions you will be asking. Consider how you would answer the following questions in both representations (in particular, how much time it would take).

Is vertex v adjacent to vertex w in an undirected graph?

What is the out-degree of a vertex v in a directed graph?

What is the in-degree of a vertex v in a directed graph?

4. Summary

- Block representation of Bintree
  - time efficient — constant time in all operations
  - 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 operations are constant time
- Tree
  - generalisation of Bintree
- Graph Representations: Adjacency List and Adjacency Matrix.