Binary Trees

The linear access time of lists makes them prohibitive for large input sets.

- Tree structures:
  - Efficient access and update to large collections of data
  - Running time of many operations is $O(\log n)$ (or based on $\log n$)
  - Some types of trees can guarantee $O(\log n)$ in worst case
  - Binary trees are widely used, relatively easy to implement

- Uses for Trees
  - Arithmetic expression evaluation
  - Storing/searching data
  - Sorting, priority queues
  - Coding, Compression
  - File systems (general trees)

- Reading: all of Ch. 5

Definitions

The definition of a binary tree is recursive:

- A binary tree is a collection of nodes.
  - The collection can be empty
  - Otherwise, a binary tree consists of
    - A distinguished node, $r$, called the root
    - Two binary trees, called the left and right subtrees, which may be empty or not

- Binary tree characteristics
  - The root of each subtree is a child of $r$
  - $r$ is the parent of each subtree root.
  - Example using recursive definition:

```
    r
   / \
  T_L  T_R
```

Definitions (cont.)

- Binary tree characteristics
  - Path from node $n_i$ to $n_k$: a sequence of nodes such that $n_i$ is the parent of $n_{i+1}$ for $1 \leq i < k$
  - Length of a path: the number of edges on the path ($k - 1$ using prior definition)
  - Parent of a node: immediate predecessor along the path from the root to that node
  - Child of a node: any immediate successor along the path from the root through that node
  - If there is a path from $n_1$ to $n_2$ then $n_1$ is an ancestor of $n_2$ and $n_2$ is a descendant of $n_1$
  - Siblings: nodes with the same parent

```
    A
   /|
  / |\n B C
  / \ / \   \
D E F
 /   /   \   \
G   H   I
```

Definitions (cont.)

- Binary tree characteristics
  - Leaf node: a node with no children
  - Internal node: a node with at least one child
  - Depth of node $n_i$: length of the unique path from the root to $n_i$
  - Height of the tree: one more than the depth of the deepest node in the tree
  - All nodes of depth $d$ are at level $d$ in the tree
  - The root is at level 0 and has depth 0

```
    A
   /|
  / |\n B D
 /   / \
C E
 /   /   \   \
F G  H  I
```
**Binary Tree Depth**

- Average depth
  - Normal binary trees: $O(\sqrt{n})$
  - Binary search trees: $O(\log n)$
- Worst-case binary tree is $O(n)$:

![Binary Tree Diagram]

**Full Binary Trees**

- **Full binary tree**: each node is either a leaf or an internal node with exactly two nonempty children
  - Example:

![Full Binary Tree Diagram]

**Complete Binary Trees**

- **Complete binary tree**:
  - If the height of the tree is $d$, then all leaves except possibly level $d$ are completely full.
  - The bottom level fills from left to right (all nodes are to the left)

![Complete Binary Tree Diagram]

**Full Binary Tree Theorem**: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes
  - Proof (by Mathematical Induction):
    - Base: a tree with one node has one leaf and no internal nodes.
    - Induction Hypothesis: Assume any FBT containing $n - 1$ internal nodes has $n$ leaves
    - Induction Step: Select an internal node whose children are both leaves and remove them...

**Corollary to FBT Theorem**: The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.
  - Proof: in an arbitrary tree, replace every empty subtree with a leaf node...
Binary Tree Node ADT

- Abstract Base Class Node:

  ```cpp
  template <class Elem>
  class BinNode {
  public:
    virtual Elem & val() = 0;
    virtual void setValue (const Elem &) = 0;
    virtual BinNode* left() const = 0;
    virtual void setLeft (BinNode*) = 0;
    virtual BinNode* right() const = 0;
    virtual void setRight (BinNode*) = 0;
    virtual bool isLeaf() = 0;
  };
  ```

Binary Tree Traversals

A traversal is the act of visiting each node in the tree in some systematic fashion.

- A traversal that lists every node exactly once is called an **enumeration**
- Each visit involves some sort of work
- Traversals are usually defined recursively
- Three types:
  - Inorder: visit left subtree, then parent, then right subtree
  - Preorder: visit parent, then both subtrees
  - Postorder: visit both subtrees, then parent
- Example:

  ```cpp
  template <class Elem>
  void preorder(BinNode<Elem>* subroot) {
    if (subroot != NULL) {
      visit(subroot);
      preorder(subroot -> left());
      preorder(subroot -> right());
    }
  }
  ```

Binary Tree Node Implementations

- Pointer-based nodes are most common
- How can we differentiate leaf and internal nodes? (And, should we?)
  - C++ Union construct
  - Use base class/subclass implementation
  - Don't
- Example: A simple node implementation (this one does not differentiate)

  ```cpp
  template <class Elem>
  class BinaryNode {
  public:
    Elem element;
    BinaryNode *left;
    BinaryNode *right;
  };
  ```

Union Implementation

- Primary problem is space inefficiency:

  ```cpp
  enum Nodetype {leaf, internal};
  class VarBinNode {
  public:
    Nodetype mytype;
    union {
      struct { // internal node
        VarBinNode *left;
        VarBinNode *right;
        operator opx;
      } int1;
      Operand var; // leaf node
    }
  };
  ```
Expression Trees

- Nodes:
  - Leaves are operands
  - Internal nodes are operators

- Expression tree for \((a + b \cdot c) + ((d \cdot e + f) \cdot g)\)

```
+------------------+
|                   |
|                   |
|                   |
|                   |
|                   |
|                   |
|                   |
|                   |
|                   |
+------------------+
      /
     /  
  a    c
      /
     /  
 b      d  
      /
     /  
 e      f
      /
     /  
 g
```

Using Inheritance (1)

- Create an abstract base class to differentiate
- Base class and Leaf node:

  ```
  class VarBinNode {
    public:
      virtual bool isLeaf() = 0;
  };

  class LeafNode : public VarBinNode {
    private:
      Operand Var;
    public:
      LeafNode(const Operand & val) { var = val; }
      bool isLeaf() { return true; }
      Operand value() { return var; }
  };
  ```

Using Inheritance (2)

- Internal Node:

  ```
  class IntNode : public VarBinNode {
    private:
      VarBinNode *left;
      VarBinNode *right;
      Operator opx;
    public:
      IntNode(const Operator op,
              VarBinNode *l, VarBinNode *r) {
        opx = op;
        left = l;
        right = r;
      }
      bool isLeaf() { return false; }
      VarBinNode *leftChild() { return left; }
      VarBinNode *rightChild() { return right; }
      Operator value() { return opx; }
  };
  ```

Constructing an Expression Tree

- Convert postfix expression to a tree
- Uses a stack to store the postfix expression
- Pseudocode algorithm:

  ```
  while (not end of postfix-expression) {
    read next symbol
    if symbol is an operand {
      create a one-node tree using operand
      push that tree onto the stack
    }
    else { // symbol is operator
      pop tree T1 from the stack
      pop tree T2 from the stack
      Form a new tree whose root is the operator
      T1 is right child
      T2 is left child
      push new tree onto stack
  ```
Using Inheritance (3)

- Composite implementation

- Base class and Leaf node:
  ```
  class VarBinNode {
  public:
    virtual bool isLeaf() = 0;
    virtual void trav() = 0;
  };
  class LeafNode : public VarBinNode {
  private:
    Operand Var;
  public:
    LeafNode(const Operand & val) {
      var = val;
    }
    bool isLeaf() { return true; }
    Operand value() { return var; }
    void trav() {
      cout << "Leaf: " << value() << endl;
    }
  };
  ```

CSC 375-Turner, Page 17

Using Inheritance (4)

- Composite Implementation

- Internal Node:
  ```
  class IntNode : public VarBinNode {
  private:
    VarBinNode *left;
    VarBinNode *right;
    Operator opz;
  public:
    IntNode(const Operator & op,
      VarBinNode *l, VarBinNode *r) {
      opx = op;
      left = l;
      right = r;
    }
    bool isLeaf() { return false; }
    VarBinNode *leftChild() { return left; }
    VarBinNode *rightChild() { return right; }
    Operator value() { return opx; }
    void trav() {
      cout << "Internal: " << value() << endl;
      if (left() != NULL) left() - > trav();
      if (right() != NULL) right() - > trav();
    }
    void traverse(VarBinNode *root) {
      if (root != NULL) root - > trav();
    }
  };
  ```

CSC 375-Turner, Page 18

Space Overhead

- FBT Theorem:
  - □ (Roughly) half the pointers are NULL
  - □ If leaves store only data, then overhead depends whether the tree is full
  - □ Example: all nodes are the same, with two pointers to children
    - Overhead fraction: \( a_f = \frac{o_f}{o} \)
    - Total space: \( t = n(2p + d) \)
    - Overhead: \( o = 2pn \)
    - If \( p = d \), then \( a_f = \frac{2p}{2p + d} = \frac{2}{3} \)
  - □ Eliminate pointers from leaf nodes:
    - Overhead
      - \( o = \frac{n}{2}(2p) \)
    - Total space
      - \( t = \frac{n}{2}(2p) + d_n \)
    - Overhead fraction
      - \( a_f = \frac{n/2(2p)}{n/2(2p) + d_n} = \frac{p}{p + d} \)
    - This is 1/2 if \( p = d \)

CSC 375-Turner, Page 19

Array Implementation

- Binary trees may be implemented with arrays
  - Structure works best for complete binary trees
  - Good example of logical vs. physical implementation
    - □ Complete binary tree is very limited
    - □ Space efficiency can be achieved

CSC 375-Turner, Page 20
Array Implementation (cont.)

- Functions that may be necessary:
  - Parent(r) = 
  - Leftchild(r) = 
  - Rightchild(r) = 
  - Leftsibling(r) = 
  - Rightsibling(r) =

Binary Search Trees

- BST property: all elements stored in the left subtree of a node whose value is k have values less than k, all elements stored in the right subtree of a node whose value is k have values ≥ k.
- The BST property allows elements to be ordered in a consistent manner
- Examples:
  - Insert 37, 24, 42, 7, 2, 40, 42, 32, 120
  - Insert 120, 42, 42, 7, 2, 32, 37, 24, 40

BST Node Class

- Code example uses friend and templates

```cpp
template <class Comparable>
class BinarySearchTree;

template <class Comparable>
class BinaryNode
{
    Comparable element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode(const Comparable &theElement, 
        BinaryNode *lt, BinaryNode *rt) 
      : element(theElement), left(lt), right(rt) { }
    friend class BinarySearchTree<Comparable>;
};
```

- Note: "Comparable" is used as a reminder
- No substantive difference from your basic binary node
- (See web site for book examples)

BST Operations

- Retrieving information:
  - find: return a pointer to that node
  - Options for failure of find:
    - throw an exception
    - return boolean as a reference or otherwise
    - return a special value
  - findMin: find the smallest element in a given (sub)tree
  - findMax: find the largest element in a given (sub)tree

- Operations that modify the tree
  - insert: insert according to BST property
  - remove: remove an element
  - removeMin: remove the smallest element (may be used by remove)
  - removeAny: remove the smallest element (used by Shaffer's dictionary ADT)

- See web site for Book's code examples
**BST Insert and Remove**

- Insert is relatively easy: preserve BST property
- Remove is the most difficult operation
- Three cases:
  - Remove a leaf
  - Remove a node that has one child
  - Remove a node that has two children
    - General strategy: replace node with smallest value of right subtree
- Examples:

**Cost of BST Operations**

- Find:
- Insert
- Remove

---

**Heaps**

- Sometimes, FIFO is not the best policy:
  - More important jobs may need to be processed first
  - Very long jobs may need to be processed last
  - Examples:
    - Print jobs
    - Multiuser OS process scheduling
- A **priority queue** can be used to satisfy this kind of environment.
  - A priority queue allows jobs to be ordered according to priority
  - Often, it supports only a small number of public operations:
    - `insert`: insert an element into the queue
    - `removelmin`: removes the minimum value
    - Alternative: `removelmax`: removes the maximum value

**Heap Concept**

A heap is a complete binary tree having the **heap property**

- Heap property:
  - In a min-heap, the value at a node is less than (or equal to) values at child nodes.
  - In a max-heap, the value at a node is greater than (or equal to) values at child nodes.
- Values in a heap are **partially ordered**
  - There is a relationship between the node and its children
  - There is no defined relationship between siblings (may be $> \geq < \leq$ in either direction)
- Heap representations are complete binary trees that (normally) use array-based implementations
Heap Operations

- Public operations that may be implemented:
  - insert: insert an element
  - removeMax or removeMin: remove the "first" element
  - remove: remove a specific element
  - isEmpty
  - isFull
  - findMax or findMin: find the "first" element
  - buildHeap: 'heapify' the contents (may be a private operation)
  - leftchild, rightchild, parent, isLeaf: standard binary tree operations

Heap ADT

- Max heap that uses the binary tree array representation

```cpp
template <Class Elem, class Comp>
class maxheap {
    private:
        Elem* heap;          // Pointer to heap array
        int n;               // Max heap size
        int size;            // Current no. elements stored
        void siftdown(int);  // Put an element in its place
    public:
        maxheap(Elem*, int, int);
        int heapsize() const;
        bool isLeaf(int pos) const;
        int leftchild(int) const;
        int rightchild(int) const;
        int parent(int) const;
        void insert(const Elem&);
        bool removeMax(Elem&);
        bool remove(int, Elem&);
        void buildheap();
};
```

Building a Heap

What is the most efficient way to build a max heap?

- Example: exchange (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6)

```
3
2 4
1 2 3
4 6 7
```

- Example: exchange (5-2), (7-3), (7-1), (6-1)

```
3
1 2
4 5 6
7
```

- It is undesirable to build a heap as you would a BST

How to Build a Max Heap

The process works in a fashion similar to an inductive proof.

- Given that \(H_1\) and \(H_2\) are already (max) heaps and \(R\) is element at root:

```
R
H_1   H_2
```

- Two possibilities exist:
  - \(R \geq \) its two children: construction is complete
  - \(R < \) one or both children: push \(R\) to its proper level as follows:
    - Exchange \(R\) with the greater-valued child
    - As long as \(R\) is out of place, descend through the tree with it until it reaches its proper place
Siftdown

- Siftdown accomplishes the “descend” process:
  ```cpp
  void maxheap::siftdown(int pos) {
    while (!isLeaf(pos)) {
      int newpos = leftChild(pos);
      int rc = rightChild(pos);
      if ((rc < n) && (heap[newpos] < heap[rc])) {
        newpos = rc;
      } else if (heap[pos] < heap[newpos]) {
        swap(heap[pos], heap[newpos]);
        pos = newpos;
      }
    }
  }
  ```

- (See web site for templated version)

- Example(s):

CSC 375-Turner, Page 33

Efficient Heap Build

For fast heap construction:

- Fill the array in input order

- Call `buildheap` procedure:
  - Works from high end of the array to low end
  - Calls `siftdown` for each item
  - Does not need to call `siftdown` for any leaf node.

- Example:
  - Input file contains 42, 21, 33, 9, 12, 6, 7, 18, 72

CSC 375-Turner, Page 34

Cost for Buildheap

- Given an unordered array, heap construction is very efficient:
  \[
  \sum_{i=1}^{\log_2 n} \frac{(i-1)n}{2^i} \approx n
  \]

- Idea:
  - Count the distance each element must go to reach final level
    - Only count downward moves
    - Once a node is processed, all nodes below it must be correct
  - Given a heap of height $d$, up to half the nodes are at depth $d$...

CSC 375-Turner, Page 35

Applications

- Selection Problem

- Event Simulation; ex: operation of a bank
  - Events:
    - Customer arrival
    - Customer departure
  - Simulation proceeds in “stages” based on events
  - Key idea is to advance the clock to next event at every stage:
    - When next customer in input file arrives
    - When a customer departs
  - Waiting line for customers is a queue
  - Waiting line for departures is a priority queue (heap)

CSC 375-Turner, Page 36
Huffman Coding Trees

Using fixed length codes can waste space

- Fixed length codes:
  - ASCII: 8 bits per character
  - Unicode: 16 bits per character

- Natural language does not have a uniform distribution of letters
  - Relative frequency of letters can be exploited

- Variable length coding:
  - Weighted path length of a leaf: weight of the leaf times its depth
  - The binary tree with minimum external path weight is the one with the minimum sum of weighted path lengths for a given set of leaves
  - Ex: a letter with high weight should have low depth to minimize its cost

CSC 375-Turner, Page 37

Huffman Tree Construction

- Create a list of nodes
  - Node contains letter/frequency pairs
  - Nodes are in increasing order of frequency
  - At each step, combine two smallest nodes into a binary tree and reorder as needed

Step 1:

Step 2:

Step 3:

CSC 375-Turner, Page 38

Huffman Tree Construction (cont.)

- Process continues until entire tree is built.

Step 4:

Step 5:

CSC 375-Turner, Page 39

Assigning Codes

- Use the completed tree
  - Right branch assigns 1 bit
  - Left branch assigns 0 bit

CSC 375-Turner, Page 40
Coding and Decoding

- A set of codes meets the **prefix property** if no code in the set is the prefix of another

- Examples:
  - Code for DEED:

  - Decode 101100110111101

  - Expected cost per letter: