CSCE 2100: Computing Foundations 1

The Tree Data Model

Tamara Schneider
Jorge Reyes-Silveyra

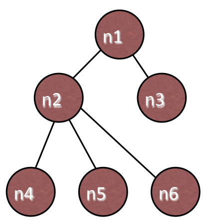
Fall 2012

Trees in Computer Science

- Data model to represent hierarchical or nested structures
  - family trees
  - charts
  - arithmetic expressions
- Certain tree types allow for faster search, insertion and deletion of elements

2

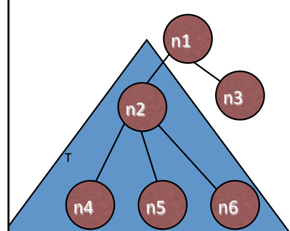
Tree Terminology [1]



- n1 is called the **root node**
- n2 and n3 are **children** of n1
- n4, n5, n6 are **children** of n2
- n4, n5, and n6 are **siblings**
- n1 is a **parent** of n2 and n3
- n3, n4, n5, and n6 are **leaves**, since they do not have any children
- All other nodes are **interior nodes**

3

Tree Terminology [2]



- n2, n3, n4, n5, and n6 are **descendants** of n1
- n1 and n2 are **ancestors** of n5
- n2 is the root of a **sub-tree** T

4

Conditions for a tree

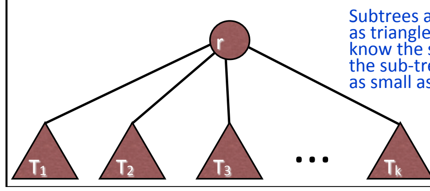
- It has a root.
- All nodes have a unique parent.
- Following parents from any node in the tree, we eventually reach the root.

5

Inductive Definition of Trees

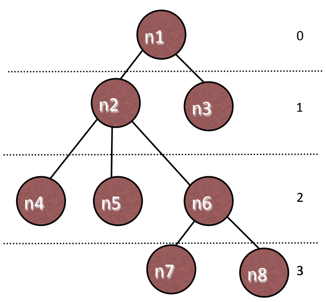
- Basis: A single node n is a tree.
- Induction: For a new node r and existing trees T_1, T_2, ... , T_k, designate r as the root node and make all T_i its children.

Subtrees are often drawn as triangles. We do not know the size of each of the sub-trees. They can be as small as a single node.



6

Height and Depth of a tree



The **height** of the tree is the length of the longest path between any node and the root.

The **depth (level)** of a node is the length of the path to the root.

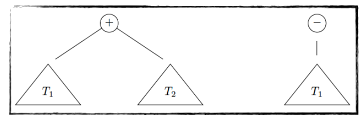
The height of the tree is 3.

The depth or level of node n5 is 2. The depth of the root node is always 0.

7

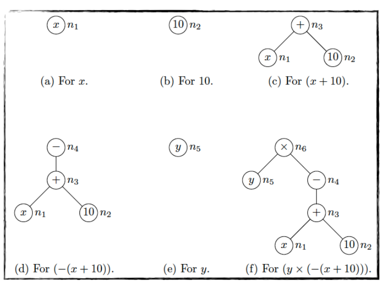
Expression Trees

- Describe arithmetic expressions
- Inductively defined
  - A tree can be as little as containing a single operand, e.g. a variable or integer (basis)
  - Trees can be inductively generated by applying the unary operator "-" to it or combing two trees via binary operators (+, - , * , /)



8

Expression Trees - Example



9

Tree Data Structures

- In C we can define a structure, similarly to linked lists.
  - use malloc to allocate memory for each node
  - nodes "float" in memory and are reached via pointers
- **In C++ we can also use classes to represent the individual nodes** (and we will for this class)
- Trees can also be represented by arrays of pointers

10

Trees as "Array of Pointers" using Classes

```
/// \file tree.h
const int MAX_CHILDREN = 10;
class CTree{
  private:
    CTree* m_pChild[MAX_CHILDREN];
    int m_nData;
  public:
    CTree(int data);
}; //CTree
```

- Each node contains data
- Each node contains an array of pointers to its children
  - Each child is represented by a tree (sub-tree)

11

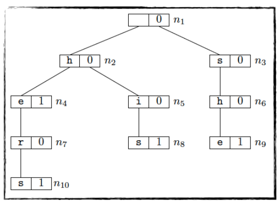
Constructor

```
/// \file tree.cpp
#include "tree.h"
CTree::CTree(int data){
  m_nData = data;
  for(int i=0; i< MAX_CHILDREN; i++)
    m_pChild[i] = NULL;
}
```

12

Trie (Prefix Tree) - Abstraction

Nodes contain "data" and a flag indicating whether a valid word ends at the node



abstract representation of a trie

he, hers, his, she

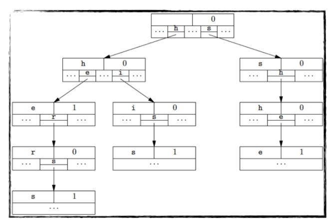
Trie Class

```
/// \file trie.h
const int MAX_CHILDREN = 10;
class CTrie{
  private:
    CTrie* m_pChild[MAX_CHILDREN];
    char m_chLetter;
    bool m_bIsWord;
  public:
    CTrie(char letter, bool isWord);
}; //CTrie
```


Trie Constructor

```
/// \file trie.cpp
#include "trie.h"
CTrie::CTrie(char letter, bool isWord) {
  m_chLetter = letter;
  m_bIsWord = isWord;
  for(int i=0; i< MAX_CHILDREN; i++)
    m_pChild[i] = NULL;
}
```


Trie (Prefix Tree) - Data Structure

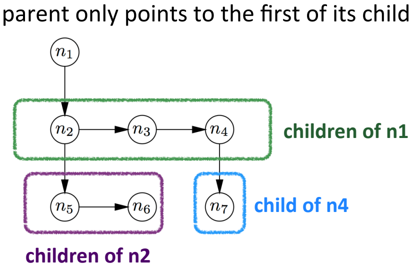


What value for MAX_CHILDREN do you expect?

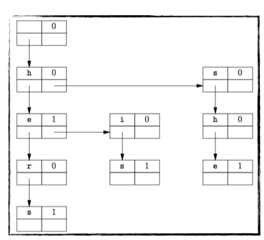
Are arrays of pointers a space efficient choice?

Leftmost-Child-Right-Sibling Abstraction

- Use a linked list instead of an array.
- A parent only points to the first of its children



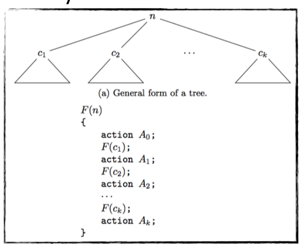
Leftmost-Child-Right-Sibling Data Structure



```
/// \file tree.h
class CTree{
  private:
    CTree* m_pLmostChild;
    CTree* m_pRSibling;
    int m_nData;
  public:
    CTree(int data);
}; //CTree
```

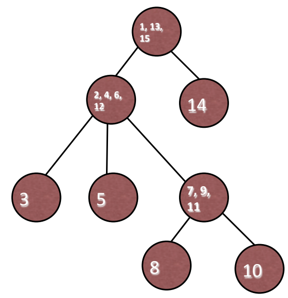
Recursion on Trees

Recursion is “natural” on trees, since trees are recursively defined.



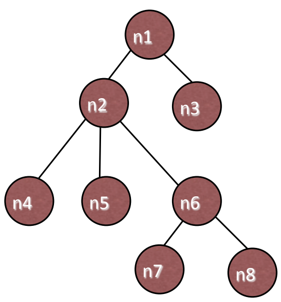
19

Order of Recursion



20

Tree Traversal: Preorder

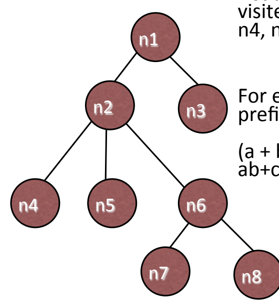


- List a node the first time it is visited
- n1, n2, n4, n5, n6, n7, n8, n3
- For expression trees: results in prefix ex-pressions, e.g.

(a + b) * c (infix)
*+abc (prefix)

21

Tree Traversal: Postorder



List a node the last time it is visited.
n4, n5, n7, n8, n6, n2, n3, n1

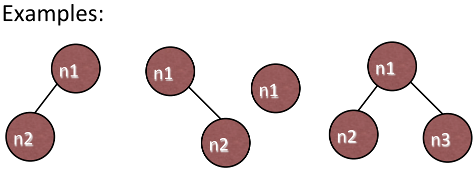
For expression trees: results in prefix expressions, e.g.

(a + b) * c (infix)
ab+c* (postfix)

22

Binary Trees

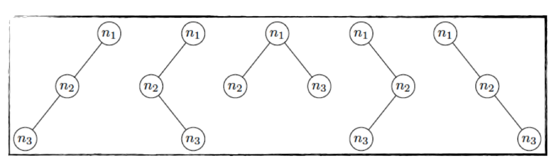
Binary trees can have **at most** 2 children. Examples:



We distinguish between the *left* and the *right* child. The distinction between them is important!

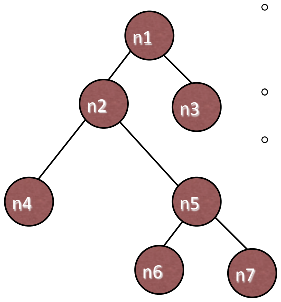
23

All binary Trees With 3 Nodes



24

Binary Tree Traversal: Inorder

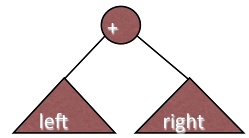


- List a node after its left child has been listed and before its right child has been listed
- n4, n2, n6, n5, n7, n1, n3
- For expression trees: results in infix expressions

25

Evaluating Expression Trees [1]

```
class CTree{
  private:
    CTree* m_pLChild;
    CTree* m_pRChild;
    char m_chOperator;
    int m_nData;
  public:
    CTree(int d, char op);
}; //CTree
```



- For interior nodes, `m_chOperator` contains an arithmetic operator (+,-,*,/)
- For leaf nodes, `m_chOperator` contains the character i for integer, and `m_nData` contains a value

26

Evaluating Expression Trees [2]

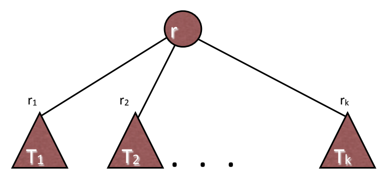
```
int CTree::eval(){
  int v1, v2;
  if(m_chOperator == 'i')return m_nData; else{
    v1 = m_pLChild->eval;
    v2 = m_pRChild->eval;
    switch(m_chOperator){
      case '+': return v1 + v2;
      case '-': return v1 - v2;
      case '*': return v1 * v2;
      case '/': return v1 / v2;
    } //switch
  } //else
} //eval
```

27

Structural Induction

Prove a statement S(T) for a tree T

- Basis: Prove the statement for a single node
- Induction: Assume the statement is true for subtrees T_1 T_2 ... T_k



28

Structural Induction - Example [1]

S(T): `T::eval()` returns the value of the arithmetic expression represented by T.

```
int CTree::eval(){
  int v1, v2;
  if(m_chOperator == 'i')return m_nData; else{
    v1 = m_pLChild->eval;
    v2 = m_pRChild->eval;
    switch(m_chOperator){
      case '+': return v1 + v2;
      case '-': return v1 - v2;
      case '*': return v1 * v2;
      case '/': return v1 / v2;
    } //switch
  } //else
} //eval
```

29

Structural Induction - Example [2]

```
int CTree::eval(){
  int v1, v2;
  if(m_chOperator == 'i')return m_nData; else{
    v1 = m_pLChild->eval;
    v2 = m_pRChild->eval;
    switch(m_chOperator){
      case '+': return v1 + v2;
      case '-': return v1 - v2;
      case '*': return v1 * v2;
      case '/': return v1 / v2;
    } //switch
  } //else
} //eval
```

Basis: T consists of a single node. `m_chOperator` has the value 'i' and the value (stored in `m_nData`) is returned.

30

Structural Induction - Example [3]

```
int CTree::eval(){
  int v1, v2;
  if(m_chOperator == 'i')
    return m_nData;
  else{
    v1 = m_pLChild->eval;
    v2 = m_pRChild->eval;
    switch(m_chOperator){
      case '+': return v1 + v2;
      case '-': return v1 - v2;
      case '*': return v1 * v2;
      case '/': return v1 / v2;
    } //switch
  } //else
} //eval
```

Induction: If T is not a leaf:

- `v1` and `v2` contain the values of the left and right subtrees (by inductive hypothesis).
- In the `switch` statement the corresponding operator is applied → correct value returned. ▮

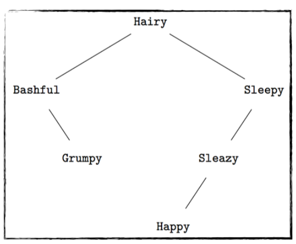
31

Binary Search Trees

- Suitable for so-called dictionary operations:
  - insert
  - delete
  - search
- Binary Search Tree property: All nodes in left subtree of a node x have labels less than the label of x, and all nodes in the right subtree of x have labels greater than the label of x.

32

Binary Search Tree - Example



Is this a valid binary search tree in lexicographic order?

33

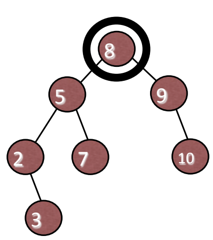
Search

Search for element x

– Check root node
  - If the root is null, return false
  - If x == root->data, return true
  - If x > root->data, search in the right subtree (recursively)
  - If x < root->data, search in the left subtree (recursively)

34

Example: Search for 7



35

Search Implementation

```
bool CTree::search(int x){
  if(x == m_nData)return true;
  if(x < m_nData){ //go left
    if(m_pLChild != NULL) //if possible
     return m_pLChild->search(x);
  }
  else //x > m_nData, go right
    if(m_pRChild != NULL) //if possible
      return m_pRChild->search(x);
  return false;
} //search
```

36

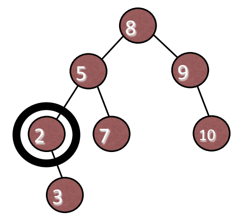
Insertion

Insert element x

- Check root node
  - If the root is null, create a new root node
  - If x == root->data, then do nothing
  - If x > root->data then insert x into the right subtree (recursively)
  - If x < root->data then insert x into the left subtree (recursively)

37

Example: Insert 8, 5, 2, 7, 9, 3, 2, 10



38

Deletion

Search for element x

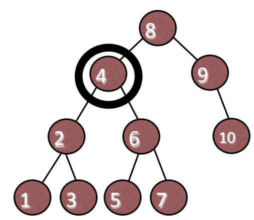
- If x does not exist, there is nothing to delete
- If x is a leaf, simply delete leaf
- If x is an interior node
  - Replace by largest element of left subtree
  - OR replace by smallest element of right subtree

Deletion is recursive! The node we use to replace the originally deleted node must be deleted recursively!

What would happen if we replaced node by the smallest element of the left subtree?

39

Example: Delete 4



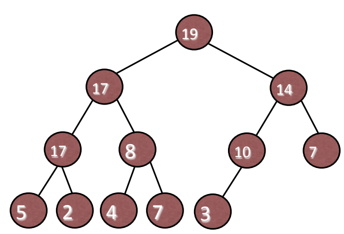
40

Priority Queues

- The elements of a priority queue have priorities. If an element with a high priority arrives, it passes all the elements with lower priorities.
  - e.g. Scheduling algorithms in operating systems make use of priority queues.
- Priority queues are often implemented using heaps, a type of partially ordered tree (POT).

41

Heaps



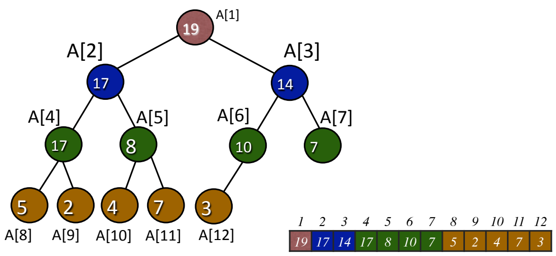
A node must have a greater value than its children.

A heap is always complete: all levels except the last level are completely filled.

Heaps are usually implemented via arrays.

42

Array Representation of Heaps

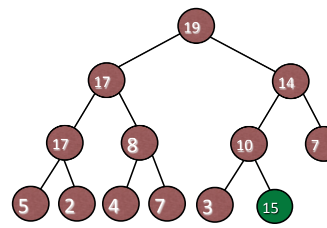


For a node A[i], we find its left child at A[2i] and A[2i+1].

Example: Children of the node A[5] are A[2*5] and A[2*5+1].

43

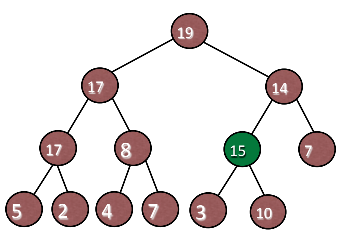
Priority Queue Operations: Insert [1]



Insert into the last level using the first available spot. If the last level is full, create a new level.

44

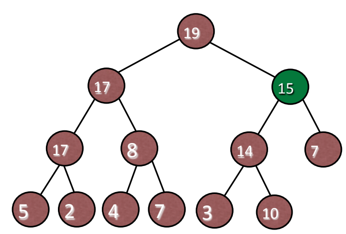
Priority Queue Operations: Insert [2]



Bubble Up: Compare with parent and exchange, if the parent is smaller.

45

Priority Queue Operations: Insert [2]

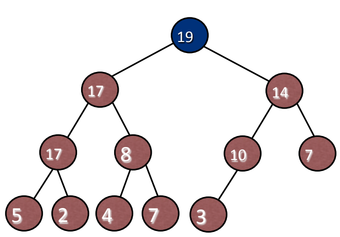


Bubble Up: Compare with parent and exchange, if the parent is smaller.

46

Priority Queue Operations: Deletemax [1]

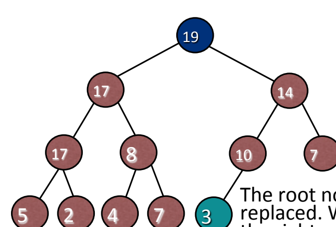
The element with the highest priority will be served first and therefore, will be removed first.



47

Priority Queue Operations: Deletemax [2]

The element with the highest priority will be served first and therefore, will be removed first.

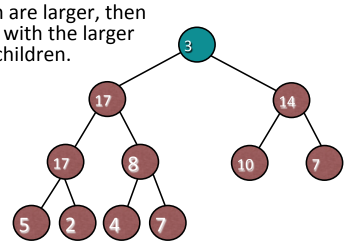


The root node must be replaced. We choose the rightmost node of the last level.

48

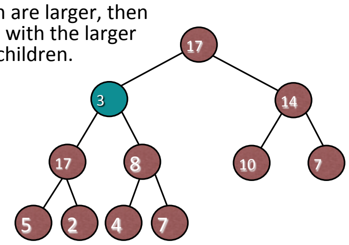
Priority Queue Operations: Deletemax [3]

<u>Bubble Down:</u> Compare with parent and if one or both of the children are larger, then exchange it with the larger one of the children.



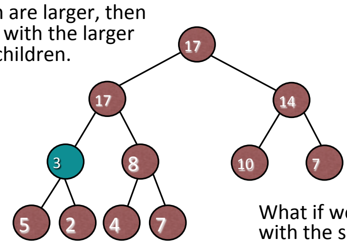
Priority Queue Operations: Deletemax [4]

<u>Bubble Down:</u> Compare with parent and if one or both of the children are larger, then exchange it with the larger one of the children.



Priority Queue Operations: Deletemax [5]

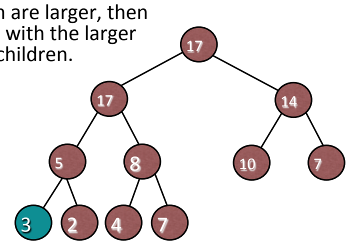
<u>Bubble Down:</u> Compare with parent and if one or both of the children are larger, then exchange it with the larger one of the children.



What if we swap it with the smaller one of the children?

Priority Queue Operations: Deletemax [6]

<u>Bubble Down:</u> Compare with parent and if one or both of the children are larger, then exchange it with the larger one of the children.

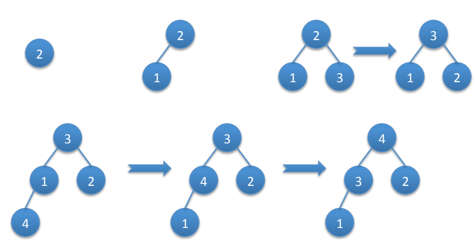


Heap Sort

- Heapify the array:
  Insert elements one by one into an initially empty MaxHeap.
- Repeatedly call deletemax:
  We obtain the elements in a sorted order from largest to smallest.
- To obtain elements sorted from smallest to largest, we can use a MinHeap instead and repeatedly call deletemin.

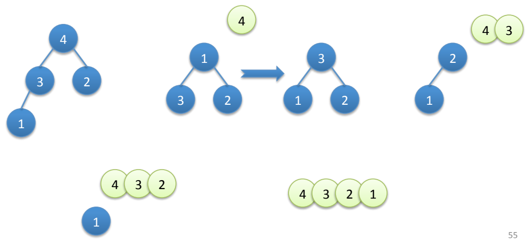
HeapSort: Example [1]

- Sort 2, 1, 3, 4
  - Insert elements into heap (Heapify)



HeapSort: Example [2]

- Sort 2, 1, 3, 4
  - Deletemax



55

Summary Heaps

- Highest priority element in the root. Each node's children are smaller than the node itself.
  - We have seen "max-heaps", where the greatest number is in the root.
  - Analogously there are "min-heap", where the smallest number is in the root.
- Insertion: Add to end and "bubble-up"
- Deletemax: Remove root element and "bubble-down"
- Heaps can be used for sorting (HeapSort)

56