Topic 5: Trees

1. General trees
2. Binary search trees
3. Tree traversal
4. BST insert and delete
5. BST implementation
6. Performance issues

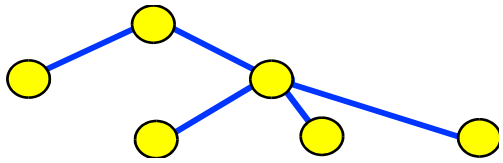
1. General trees

- A tree represents a hierarchy
- Applications:
  - disk directory
  - arithmetic expressions
  - parsing
- Access time and height of tree
- Implementations:
  - *m*-ary tree
  - tree using child and sibling links

A tree is hierarchic

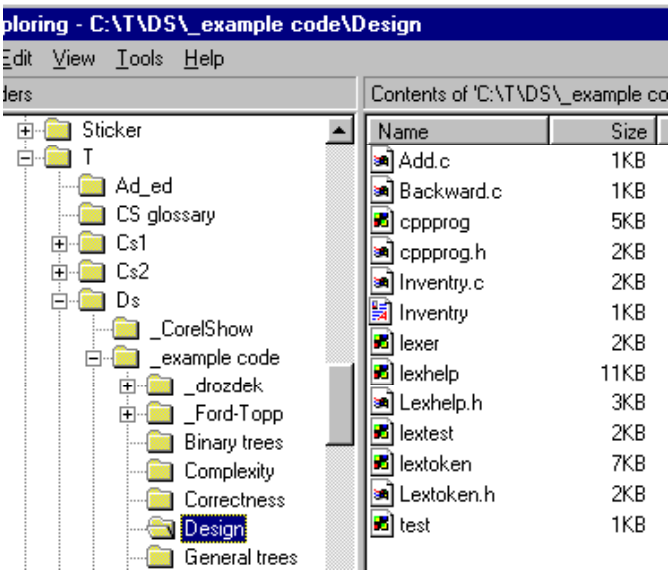
- Examples:
  - Family tree
  - Organizational hierarchy
  - Module hierarchy chart
  - Taxonomy
  - Parse tree
  - Disk directory
- A tree has a *root* at top, *leaves* at bottom
- A nonroot node has one *parent* node
- A nonleaf has one or more *child* nodes
- Every tree node is the root of a subtree

Tree: a connected acyclic graph

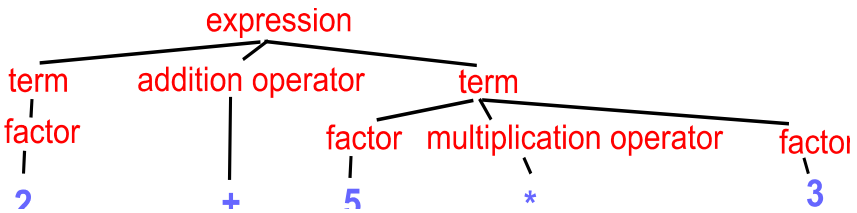


- Exactly one path joins any pair of vertices
- Removing an edge disconnects the tree
- Adding any edge creates a cycle
- Cardinalities of edges and vertices:
  $|E| = |V| - 1$

Application: disk directories



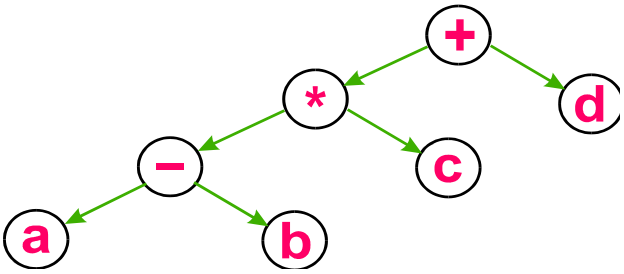
Parse trees



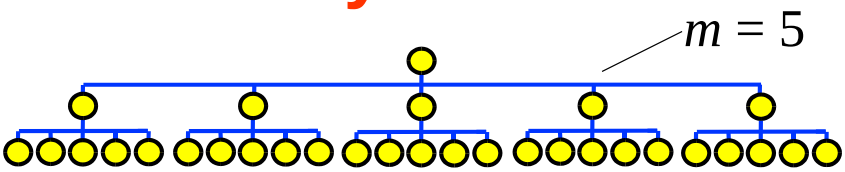
- Compilers build a program structure from tokens
- Root may be *program*
- Lexemes are leaves of tree
- Nonterminal syntax elements (e.g., *expression*, *factor*) are internal tree nodes or the root

Expression tree

- Each operator is the root of a subtree
- Example: (a – b) * c + d

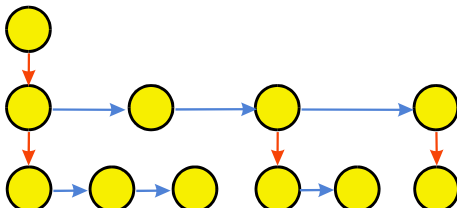


m-ary tree



- Advantage: *very* fast access
- Disadvantage: high overhead, large waste of unused links
- Disadvantage: predetermined limit on number of a node's children

General tree node structure



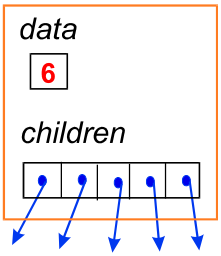
- Data item
- Reference to child
- Reference to sibling

Design concept: A node has one child reference. It points to the first node of a linked list of the node's children

Implementations of general trees

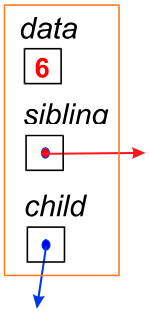
(a) *m*-ary tree:

- Each node has 0 to *m* children
- Each child has one pointer to it



(b) Alternate implementation without limit on children per node:

- Each node has one child link and one sibling link
- Each node may thus be a parent
- Each node may also be part of a list of siblings



Classes for a binary search tree

```
class nodes {
public:
  nodes()
  { *spelling = '\0';   left = right = NULL; }
  friend class word_trees;
private:
  char spelling[40];
  nodes *left,*right;
};
```

Node type

BST class

```
class word_trees {
public:
      word_trees() { root = NULL; }
      nodes *get_root() { return root; }
      void retrieve(char *filenm);
      void display(nodes *subtree);
      void insert(char *s,nodes *&nd);
      nodes *new_node(char *s);
      bool search(char* key);
private:
      nodes *root;
};
```

[*treesort.cpp*]

Constructor, destructor for BST nodes

```
nodes::nodes(char* s)
{
   strcpy(spelling, s);
   left = right = NULL;
}

nodes::~nodes()
{
   delete left;
   delete right;
}
```

Given this destructor, a tree may be deallocated automatically by deallocating the header

BST node insertion

```
void word_trees::insert(char *s,nodes
*&nd)
{
  if (nd != NULL)
    if (strcmp(s,nd->spelling) > 0)
      insert(s,nd->right); // right child
    else  // <s> not too large
      insert(s,nd->left);  // left child
  else // leaf
    nd = new nodes(s);
}
```

Reference to a pointer

This function creates a node with string **s**, *inserts it into tree below node* **nd**.

[*treesort.cpp*]

Binary-tree search

```
bool word_trees::search(char *key,nodes *nd)
// Tells if <key> in subtree with root <nd>.
{
  if (nd == NULL) return false;
  else
  {
    int strdiff = strcmp(key,nd->spelling);
    if (strdiff == 0)
      return true;
    else
      if (strdiff < 0)
        return search(s,nd->left);
      else
        return search(s,nd->right);
  }
}
```

Inorder traversal for output

```
void word_trees::display(nodes *subtree)
// Prints subtree thruough inorder
traversal.
{
   if (subtree != NULL)
   {
      display (subtree->left);
      cout << subtree->spelling << " ";
      display (subtree->right);
   }
}
```

- How many recursive calls?
- For *d* levels of recursion, what is running time?

BST-clear(root)

Postcondition:
Subtree with root *root* is empty

if root ≠ null
  BST-clear (*left* (*root*))
  BST-clear (*right* (*root*))
  Deallocate *root*
  root ← null

- This is a postorder traversal

BST-sorting a file of integers

```
#include "bst.h"

void main()
{
   BST<int> tree;
   cout << "\nText of file INTSORT.DAT: \n";
   tree.retrieve("INTSORT.DAT");
   cout << "\nText of same file, sorted:\n";
   tree.display(tree.get_root());
}
```

Tree *is a binary search tree of* **int**s

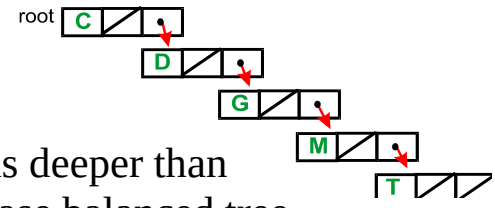
[*intsort.cpp*]

Output:

```
Text of file INTSORT.DAT:
83 61 3 31 37 96 28 15 40 62 53 89 58 18
Text of same file, sorted:
3 15 18 28 31 37 40 53 58 61 62 83 89 96
```


6. Performance issues

Worst-case degenerate tree



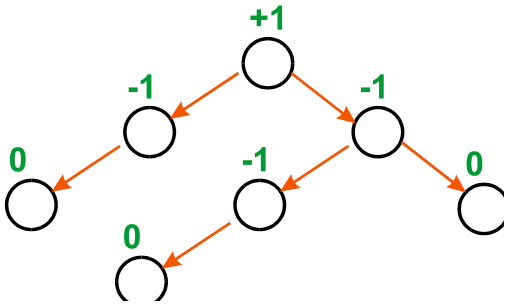
- Imbalanced tree is deeper than best or average-case balanced tree
- Access time: O(____)
- How much worse is this running time than for balanced BST?

Balanced BSTs are faster

- Very long paths from root to nodes can be avoided if all leaves are near same level
- A *balanced* tree is one in which the levels of any two leaves differ by 1, at most
- Algorithms exist to balance BSTs by rotating the tree around the root

AVL tree:

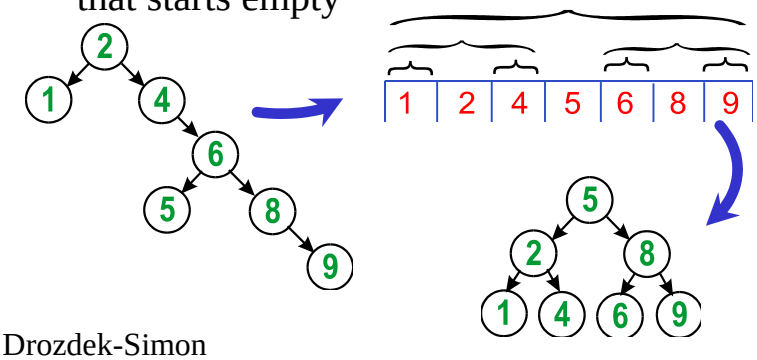
A BST in which the difference in heights for left and right subtrees is no greater than 1, for all nodes



For each node we may calculate the difference between heights of left, right subtrees

Tree balancing technique based on binary search

1. Sort data
2. Recursively insert median values into a tree that starts empty



Drozdek-Simon

Build-balanced (*BST*, *A*, *first*, *last*)

If *first* ≤ *last*

mid ← ⌊(*first* + *last*) ÷ 2⌋

BST-insert (*BST*, *A*[*mid*])

Build-balanced (*BST*, *A*, *first*, *mid*-1)

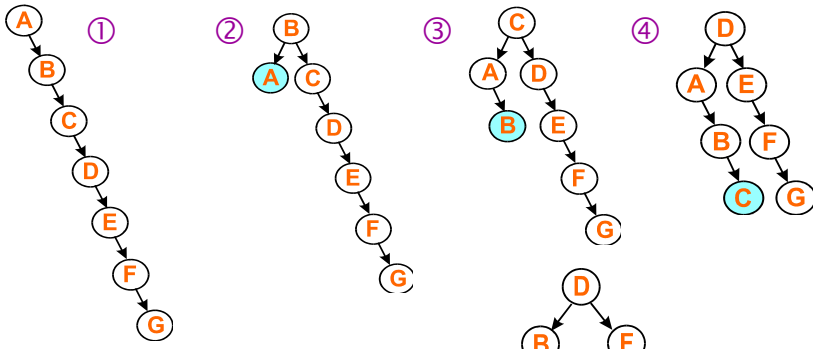
Build-balanced (*BST*, *A*, *mid*+1, *last*)

- *Preconditions: BST* empty, *A* is array sorted ascending, *first* and *last* are subscripts to *A*
- *Postconditions: BST* contains all elements of *A*, *BST* is balanced

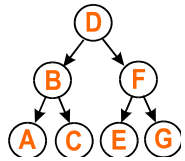
[*See Drozdek and Simon, p. 178*]

Tree balancing by delete-insert?

- *Intuition:* Starting with a degenerate tree, delete-and-insert the root $n \div 2$ times, then recurse for the root's degenerate subtrees



⑤ *Repeat steps 2-4 for the subtrees A-B-C and E-F-G*



<u>DI-balance(*BST*, *root*, *n*)</u>

If *root* is not null
  halfway ← ⌊*n* ÷ 2⌋
  for *i* ← 1 to *halfway*
    d ← *data*(*root*)
    BST-delete(*root*, *BST*)
    BST-insert(*d*, *BST*)
  DI-balance(*BST*, *left*(*root*), *halfway*)
  DI-balance(*BST*, *right*(*root*), *halfway*)

- *Preconditions*: *BST* is a binary search tree, *root* is root of a degenerate subtree in *BST* (contains no left child nodes) [*D. Keil 1/99*]
- *Postcondition: BST* is balanced, has same data as before

Build-degenerate (*source*, *dest*)

If *source* not null
  Build-degenerate (*left*(*source*), *dest*)
  BST-insert (*data*(*source*), *dest*)
  Build-degenerate (*right*(*source*), *dest*)

- *Preconditions*: *source* is root of a BST, *dest* is an empty BST
- *Postcondition: dest* is a degenerate (no left-child nodes) copy of *source*
- *Challenges:* (1) Does this work?
  (2) Does *DI-balance* work?
  (3) What are running times?

[*D. Keil 1/99*]

BST discussion questions

- How should we handle the situation where a value is submitted for insertion into a BST and the tree already contains the value?
- What are the costs and benefits of using a BST to implement a collection, versus an array or vector; a linked list
- Would a BST make a good
  - stack
  - queue
  - priority queue

Terms

General trees
child
depth of a node
expression tree
family tree
general tree
height of a tree
hierarchy
leaf
level
m-ary tree
parent
root
subtree
tree node

Binary search trees
AVL tree
balanced tree
binary search tree
BST property
BST deletion
BST insertion
BST search
BST node
degenerate tree
inorder traversal
postorder traversal
preorder traversal

References

A. Drozdek, P. Simon. *Data Structures in C*. PWS, 1995.