CH7. TREES

CSED233 Data Structure
Prof. Hwanjo Yu
POSTECH

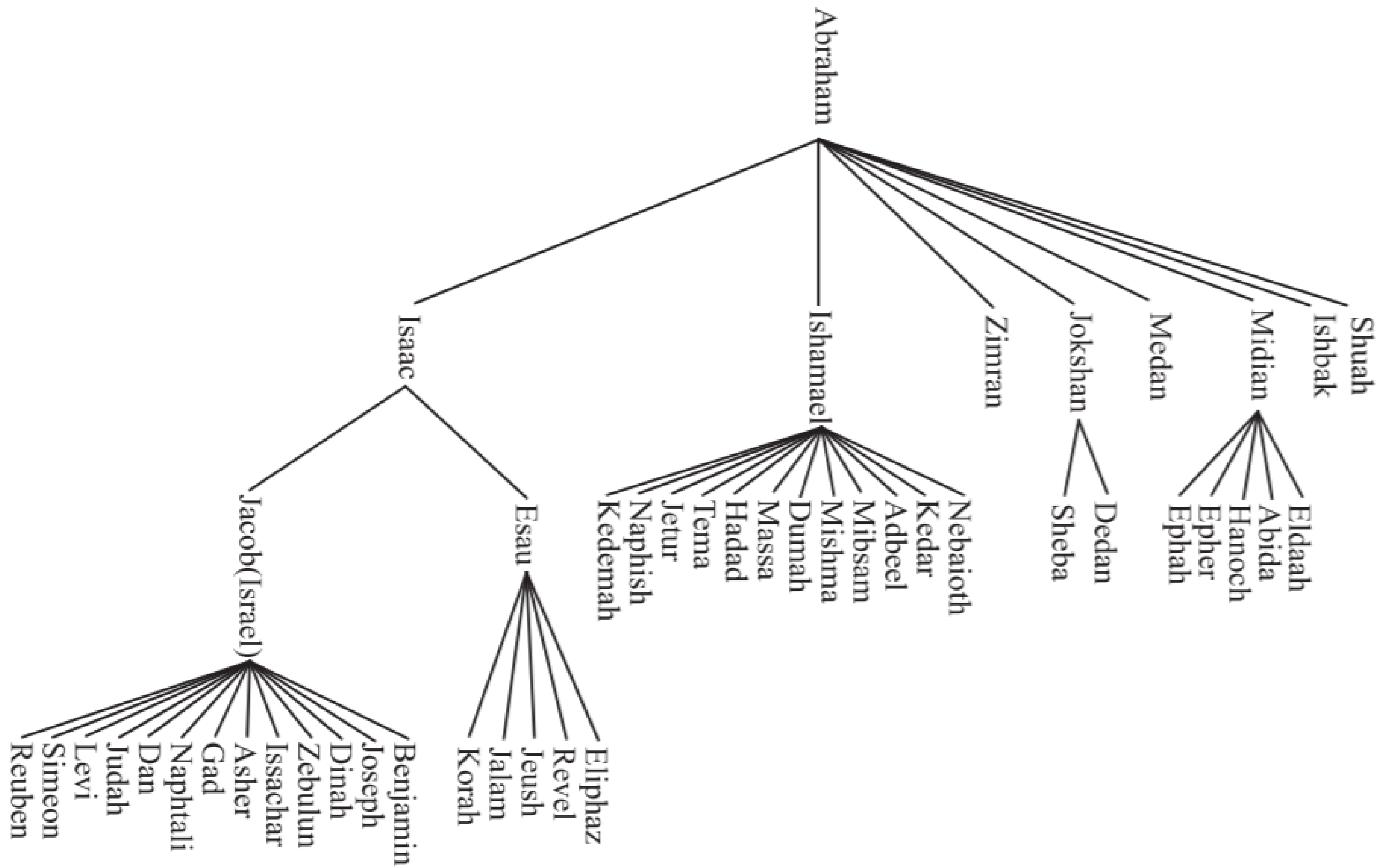
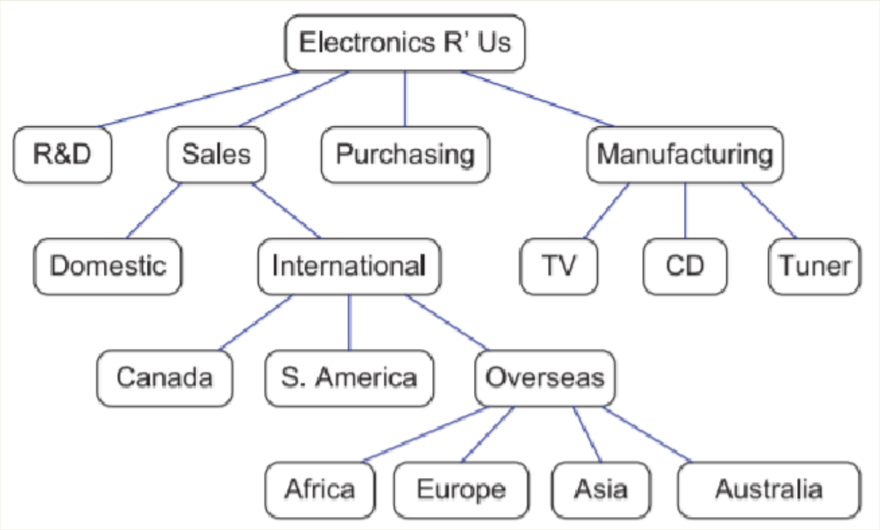


Figure 7.1: A family tree showing some descendants of Abraham, as recorded in Genesis, chapters 25–36.

Tree Definitions and Properties



- Tree
  - *Parent*
  - *Children*
  - *Root*
  - *Siblings*
  - *External nodes (leaves)*
  - *Internal nodes*
  - *Ancestor*
  - *Descendent*
  - *Subtree*
  - *Edge and path*
- Position
  - *p.parent()*
  - *p.children(): returns a position list*
  - *root(): returns a position*
  - *positions(): returns a position list of all the nodes*

C++ Tree Interface

```
template <typename E>                // base element type
class Position<E> {                  // a node position
public:
  E& operator*();                    // get element
  Position parent() const;           // get parent
  PositionList children() const;     // get node's children
  bool isRoot() const;               // root node?
  bool isExternal() const;           // external node?
};
```

Code Fragment 7.1: An informal interface for a position in a tree (not a complete C++ class).

```
template <typename E>                // base element type
class Tree<E> {
public:                              // public types
  class Position;                    // a node position
  class PositionList;                // a list of positions
public:                              // public functions
  int size() const;                  // number of nodes
  bool empty() const;                // is tree empty?
  Position root() const;             // get the root
  PositionList positions() const;    // get positions of all nodes
};
```

Code Fragment 7.2: An informal interface for the tree ADT (not a complete class).

- PositionList
  - *An STL list of Position*
  - *“std::list<Position>”*

A Linked Structure for General Trees

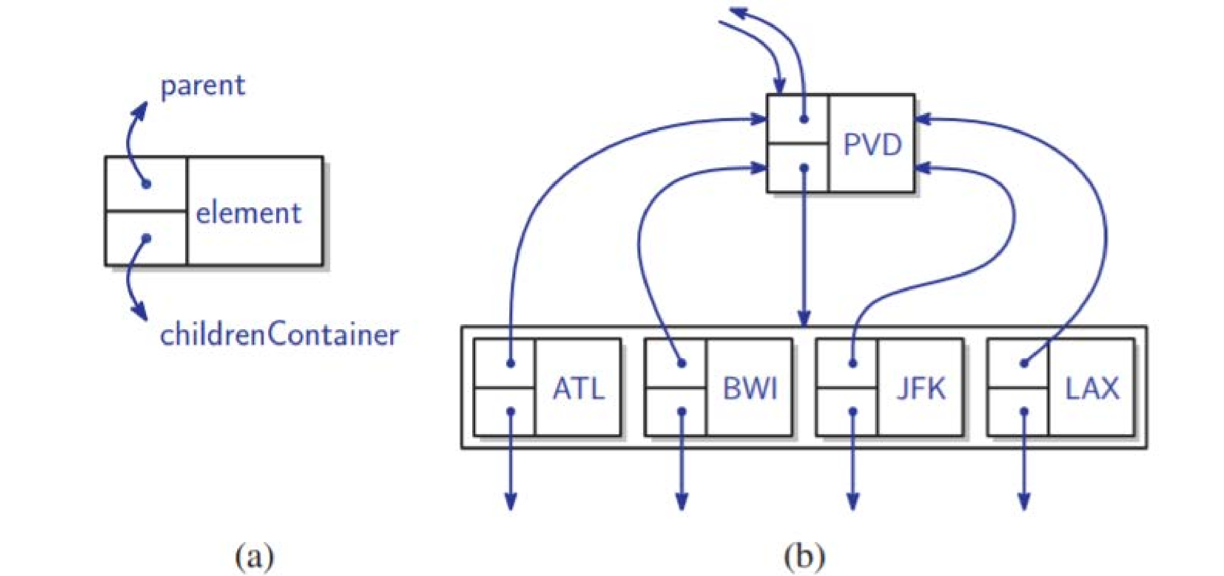


Figure 7.5: The linked structure for a general tree: (a) the node structure; (b) the portion of the data structure associated with a node and its children.

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

Depth and Height

```
int depth(const Tree& T, const Position& p) {
  if (p.isRoot())
    return 0;                              // root has depth 0
  else
    return 1 + depth(T, p.parent());       // 1 + (depth of parent)
}
```

```
int height1(const Tree& T) {
  int h = 0;
  PositionList nodes = T.positions();          // list of all nodes
  for (Iterator q = nodes.begin(); q != nodes.end(); ++q) {
    if (q->isExternal())
      h = max(h, depth(T, *q));                // get max depth among leaves
  }
  return h;
}
```

```
int height2(const Tree& T, const Position& p) {
  if (p.isExternal()) return 0;                // leaf has height 0
  int h = 0;
  PositionList ch = p.children();              // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q)
    h = max(h, height2(T, *q));
  return 1 + h;                                // 1 + max height of children
}
```

Preorder Traversal

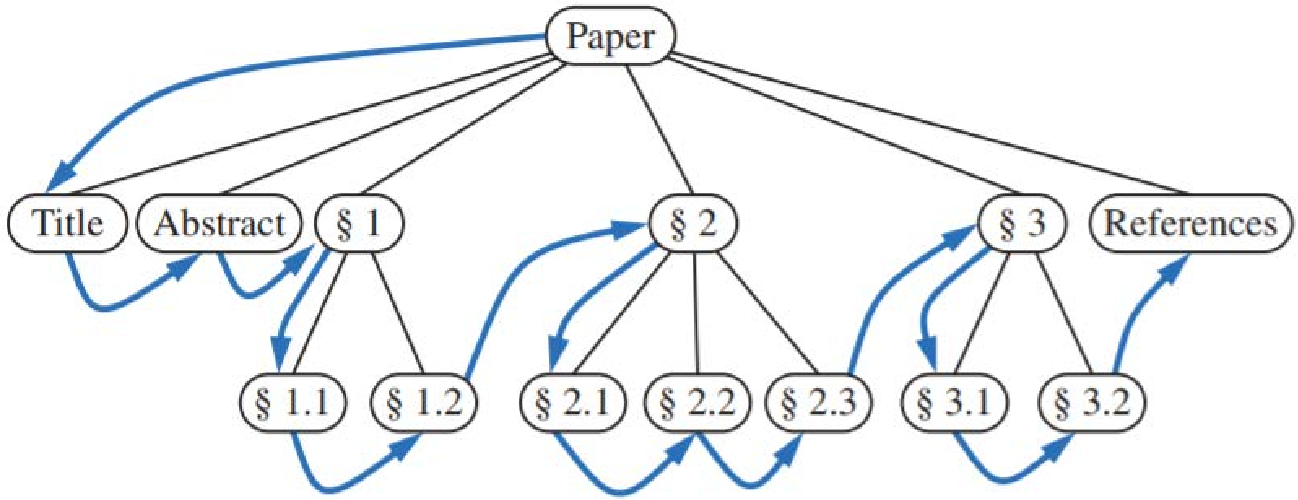


Figure 7.6: Preorder traversal of an ordered tree, where the children of each node are ordered from left to right.

```
void preorderPrint(const Tree& T, const Position& p) {
  cout << *p;                                    // print element
  PositionList ch = p.children();                // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    cout << " ";
    preorderPrint(T, *q);
  }
}
```

Code Fragment 7.10: Method preorderPrint(T, p) that performs a preorder printing of the elements in the subtree associated with position p of T.

Postorder Traversal

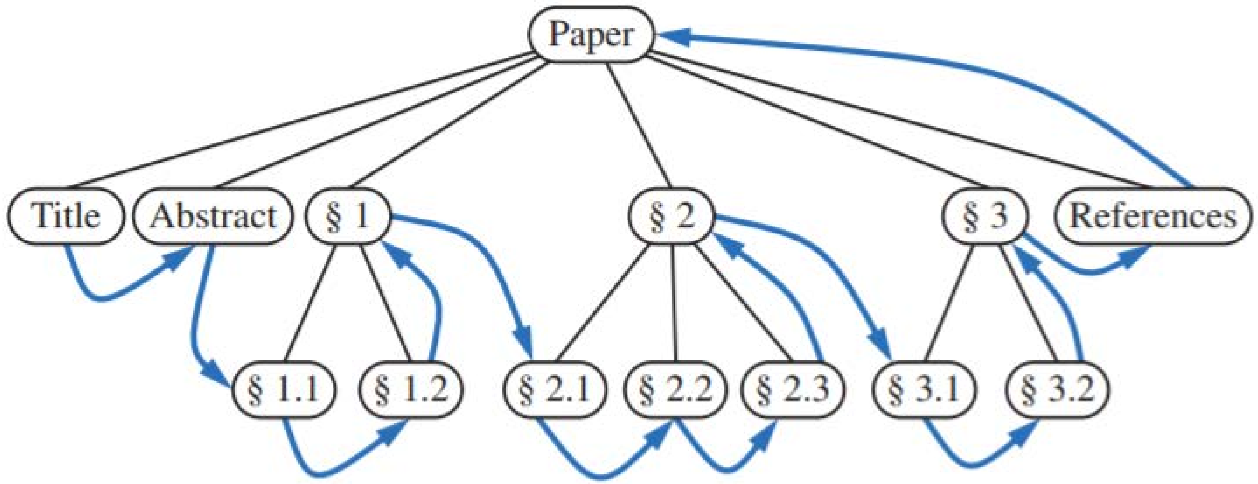


Figure 7.8: Postorder traversal of the ordered tree of Figure 7.6.

```
void postorderPrint(const Tree& T, const Position& p) {
  PositionList ch = p.children();                    // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    postorderPrint(T, *q);
    cout << " ";
  }
  cout << *p;                                        // print element
}
```

Code Fragment 7.13: The function postorderPrint(T, p), which prints the elements of the subtree of position p of T.

Postorder Traversal

```
int diskSpace(const Tree& T, const Position& p) {
  int s = size(p);                             // start with size of p
  if (!p.isExternal()) {                       // if p is internal
    PositionList ch = p.children();            // list of p's children
    for (Iterator q = ch.begin(); q != ch.end(); ++q)
      s += diskSpace(T, *q);                   // sum the space of subtrees
    cout << name(p) << ": " << s << endl; // print summary
  }
  return s;
}
```

Code Fragment 7.14: The function diskSpace, which prints the name and disk space used by the directory associated with p, for each internal node p of a file-system tree T. This function calls the auxiliary functions name and size, which should be defined to return the name and size of the file/directory associated with a node.

Other Traversal

- Depth-First Traversal
  - *Preorder: me first and children later*
  - *Postorder: children first and me later*
  - *Use Stack (or recursive function)*
- Breadth-First Traversal
  - *Use Queue*

Binary Trees (§ 7.3)

- Binary trees
  - *Every node has at most two children*
  - *Left child and right child*
  - *A binary tree is* ***proper*** *if each node has either zero or two children (**full** binary tree)*

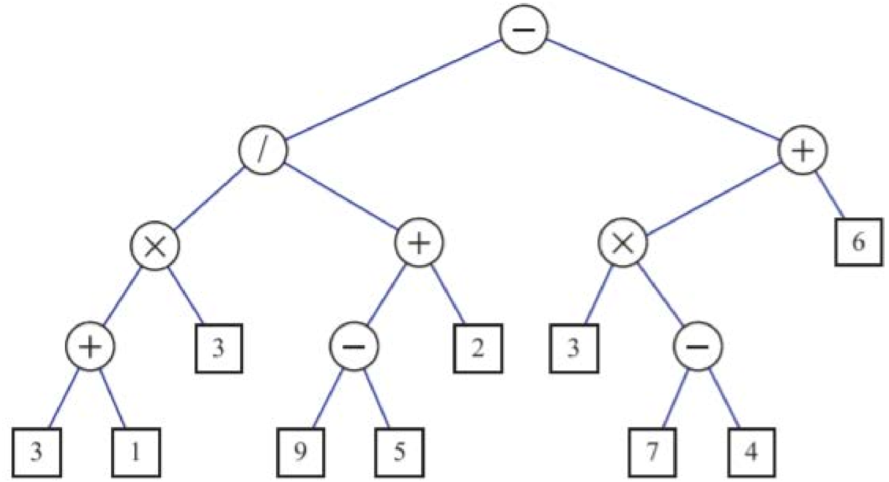
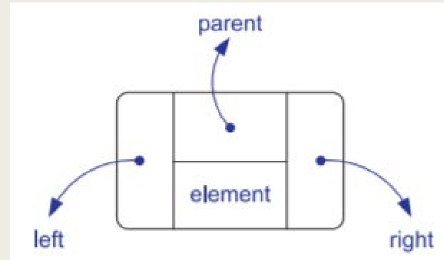


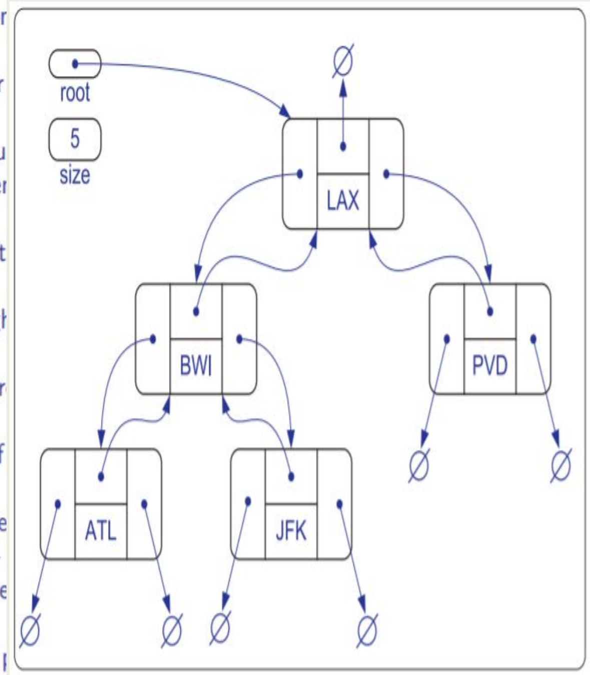
Figure 7.11: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

The Binary Tree ADT

```
struct Node {                                              // a no
  Elem   elt;                                              // elem
  Node*  par;                                              // parer
  Node*  left;                                             // left c
  Node*  right;                                            // right
  Node() : elt(), par(NULL), left(NULL), right(NULL) { }
};
```




```
class Position {                                  // positior
private:
  Node* v;                                        // pointer
public:
  Position(Node* _v = NULL) : v(_v) { }           // constru
  Elem& operator*()                               // get eler
    { return v->elt; }
  Position left() const                           // get left
    { return Position(v->left); }
  Position right() const                          // get righ
    { return Position(v->right); }
  Position parent() const                         // get par
    { return Position(v->par); }
  bool isRoot() const                             // root of
    { return v->par == NULL; }
  bool isExternal() const                         // an exte
    { return v->left == NULL && v->right == NULL; }
  friend class LinkedBinaryTree;                  // give tre
};
typedef std::list<Position> PositionList;         // list of p
```




```
typedef int Elem;
class LinkedBinaryTree {
protected:
  // insert Node declaration here...
public:
  // insert Position declaration here...
public:
  LinkedBinaryTree();
  int size() const;
  bool empty() const;
  Position root() const;
  PositionList positions() const;
  void addRoot();
  void expandExternal(const Position& p);
  Position removeAboveExternal(const Position& p);
  // housekeeping functions omitted...
protected:
  void preorder(Node* v, PositionList& pl) const;
private:
  Node* _root;
  int n;
};
```

Binary Tree Update Functions

expandExternal(p): Transform p from an external node into an internal node by creating two new external nodes and making them the left and right children of p, respectively; an error condition occurs if p is an internal node.

removeAboveExternal(p): Remove the external node p together with its parent q, replacing q with the sibling of p (see Figure 7.15, where p's node is w and q's node is v); an error condition occurs if p is an internal node or p is the root.

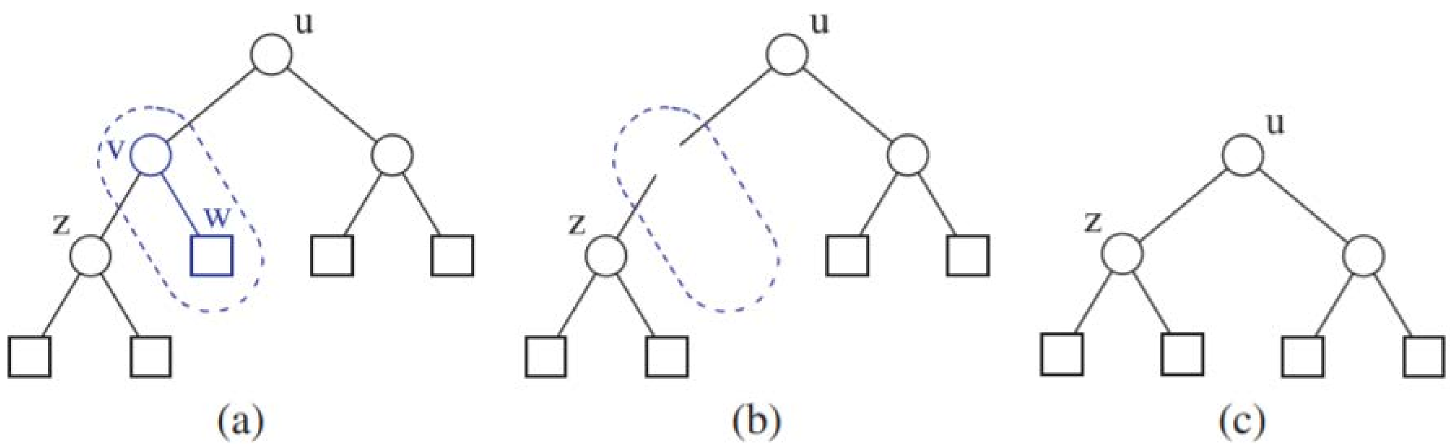


Figure 7.15: Operation removeAboveExternal(p), which removes the external node w to which p refers and its parent node v.

Binary Tree Update Functions

```
void LinkedBinaryTree::expandExternal(const Position& p) {
  Node* v = p.v;                              // p's node
  v->left = new Node;                         // add a new left child
  v->left->par = v;                           // v is its parent
  v->right = new Node;                        // and a new right child
  v->right->par = v;                          // v is its parent
  n += 2;                                     // two more nodes
}
```

```
LinkedBinaryTree::Position                    // remove p and parent
LinkedBinaryTree::removeAboveExternal(const Position& p) {
  Node* w = p.v; Node* v = w->par;            // get p's node and parent
  Node* sib = (w == v->left ? v->right : v->left);
  if (v == _root) {                           // child of root?
    _root = sib;                              // ...make sibling root
    sib->par = NULL;
  }
  else {
    Node* gpar = v->par;                      // w's grandparent
    if (v == gpar->left) gpar->left = sib;    // replace parent by sib
    else gpar->right = sib;
    sib->par = gpar;
  }
  delete w; delete v;                         // delete removed nodes
  n -= 2;                                     // two fewer nodes
  return Position(sib);
}
```

Binary Tree Traversals

```
                                                    // list of all nodes
LinkedBinaryTree::PositionList LinkedBinaryTree::positions() const {
  PositionList pl;
  preorder(_root, pl);                              // preorder traversal
  return PositionList(pl);                          // return resulting list
}
                                                    // preorder traversal
void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const {
  pl.push_back(Position(v));                        // add this node
  if (v->left != NULL)                              // traverse left subtree
    preorder(v->left, pl);
  if (v->right != NULL)                             // traverse right subtree
    preorder(v->right, pl);
}
```

Algorithm inorder(T, p):
  if p is an internal node **then**
    inorder(T, p.left())   {recursively traverse left subtree}
  perform the "visit" action for node p
  if p is an internal node **then**
    inorder(T, p.right())   {recursively traverse right subtree}

Code Fragment 7.27: Algorithm inorder for performing the inorder traversal of the subtree of a binary tree T rooted at a node p.

Binary Search Trees

- Binary search trees
  - *Internal node p stores an element $x(p)$*
  - $x(p_{left}) \leq x(p) \leq x(p_{right})$
  - *No elements in external nodes*
  - *Inorder traversal visits the elements in nondecreasing order*

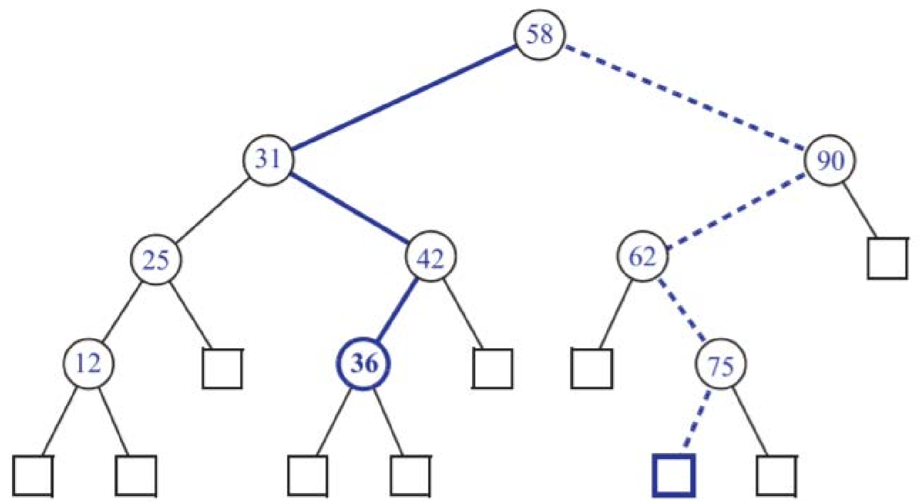


Figure 7.19: A binary search tree storing integers. The blue solid path is traversed when searching (successfully) for 36. The blue dashed path is traversed when searching (unsuccessfully) for 70.