Tree traversals

- Systematic visiting of all nodes in the tree.
- Each node visited once, does not specify order.
- Breadth-first traversal: visit each node, starting from the lowest level and moving down by level, visiting nodes from left to right.
- Depth-first traversals: go in subtree as deep as you can, backtrack. Differ on the order of visiting root, left, right subtrees.
  - preorder: VLR.
  - inorder: LVR.
  - postorder: LRV.
- these definitions are recursive.
void BST::breadthFirst()
{
    Queue<BSTNode> q;
    BSTNode *p = root;
    if (p != 0){
        q.enqueue(p);
        while (!q.empty()){
            p = q.dequeue();
            visit(p);
            if (p->left != 0)
                q.enqueue(p->left);
            if (p->right != 0)
                q.enqueue(p->right);
        }
    }
}
DFS traversals of a tree

```cpp
void BST::inorder(BSTNode * p)
{
    if (p!=0)
    {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}
void BST::preorder(BSTNode * p)
{
    if (p!=0)
    {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}
void BST::postorder(BSTNode * p)
{
    if (p!=0)
    {
        postorder(p->left);
        postorder(p->right);
    }
}
```
Preorder traversal
Inorder traversal
Postorder traversal
Intemezzo: Recursion

- One of the basic rules: objects/concepts in terms of simpler objects/concepts.
- However: many programming concepts "define themselves". recursive definitions.
- A recursive definition consists of two parts: anchor(ground) case, rules for construction of objects out of basic elements/objects already constructed.

**Example:** natural numbers.

(i) \(0 \in \mathbb{N} \).

(ii) \((x \in \mathbb{N}) \implies (x + 1 \in \mathbb{N})\).

(iii) these are all natural numbers.

**Example:** natural numbers in base 10.

(i) \(0, 1, 2, \ldots, 9 \in \mathbb{N} \).

(ii) \((x \in \mathbb{N}) \implies (x0, x1, \ldots, x9 \in \mathbb{N})\).

(iii) these are all natural numbers.

**Example:** factorial.

\[
n! = \begin{cases} 
1 & \text{if } n = 0 \\
n \cdot (n - 1)! & \text{if } n \geq 1,
\end{cases}
\]
Function calls and recursive implementation

```c
unsigned int factorial(unsigned int n)
{
    if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

- What happens when you call function?
- If function has formal parameters, they have to be initialized to the values passed as actual parameters.
- System has to know where to resume execution after function has finished.
- System has to store context of the call.
- Variable `x` might exist in both called context and calling context.
• Stack frame (activation record): data area containing this information.
• values for all parameters of the function, address of the first entry in an array (if passed).
• Local variables: values can be stored elsewhere, descriptor, pointer to locations where they are stored.
• Dynamic link, pointer to caller’s activation record
• Return address to resume control by the caller, the address of the caller’s instruction immediately following the call.
• Return value for a function not declared as void.
Example: stack frame

void main(int argc, char *argv[])
{
    int x = 9;
    .......
    f1(x);
    .......
}

void f1(int i)
{
    int x;
    x = 7;
    .......
    f20;
    .......
}

void f20
{
    int x;
    .......
    control transferred to BM
    give BM the current PC, SP
    .......
}
Anatomy of recursive call

double power(double x, unsigned int n) {
    if (n==0)
        return 1.0;
    return x * power(x, n-1);
}

power(x, 4)
    power(x, 3)
        power(x, 2)
            power(x, 1)
                power(x, 0)
                    1
                        x
                            x
                                x
                                    x
                                        x
Non-recursive implementation of power

double nonRecPower(double x, unsigned int n) {
    double result = 1;
    for(result = x; n > 1; --n) {
        result *= x;
    }
    return result;
}

• Recursion: more intuitive.
• Shorter than the iterative version.
• Costlier than the iterative version.
Excessive recursion

- Recursion is logically simple and yields readable code, but has high overhead (stack).
- Can sometimes overflow the stack.
- Many times nonoptimal.
- Example: Fibonacci numbers.

\[
\text{fib}\left(n\right) = \begin{cases} 
1 & \text{if } n = 0, 1 \\
\text{fib}(n - 1) + \text{fib}(n - 2) & \text{otherwise.}
\end{cases}
\]

- Recursive implementation: immediate.
- Fib(6) calls fib(5) and fib(4). Fib(5) also calls fib(4). Different stack frames, so different computations!
- Exponential number of calls to fib.
Good case: Tail recursion

void tail(int i){
    if (i>0){
        cout << i<< " ";
        tail(i-1);
    }
}

void iterativeEquivalentOfTail(int i){
    for( ; i>0; i--)
        cout << i<< " ";
}

• Function: recursive call at the end.
• Basically a loop.
• Tail recursion: can be replaced with iteration.
void reverse()
{
    char ch;
    cin.get(ch);
    if (ch != 'n'){
        reverse();
        /* 204 */ cin.put(ch);
    }
}

• main calls reverse() with parameter "ABC".
• an activation record created for parameter ch and return address. Not for the result since the function return type is void.
• stack frame: ('a', (to main))-(>'b',(204),'a',(to main))-(>'c',(204),'b',(204),'a',(to main))-(>'n',(204),'c',(204),'b',(204),'a',(to main)).
void iterativeReverse()
{
    char stack[80];
    register int top = 0;
    cin.get(stack[top]);
    while(stack[top] != \n)
        cin.get(stack[++top]);
    for (top -=2; top >=0; cout.put(stack[top--]));
}
**Nonrecursive implementation: comments**

- Name stack for array not accidental. Our stack takes over the run-time stack’s duty.
- The transformation of nontail recursion into tail recursion explicitly involves handling a stack.
Indirect recursion

- Preceding slides: $f$ calls itself. However, $f$ can call itself indirectly, via chain of other functions. Chain can have arbitrary length e.g. $f() - > f_1() - > \ldots - > f_n() - > f()$. Also: $f$ can call itself through different chains.
- E.g. receive()-&gt;decode()-&gt;store()-&gt;receive()-&gt;decode()-&gt;store()-&gt;receive()-&gt;decode()-&gt;store()-&gt;...

```
receive(buffer)
    while(buffer is not filled up)
        if information still incoming
            get a character and store it in the buffer
        else exit()
    decode(buffer);

decode(buffer)
    decode information in buffer;
    store(buffer);

store(buffer)
    transfer information from buffer to file;
    receive(buffer);
```
Nested recursion

- More complicated case: function not only defined in terms of itself, but used as a parameter.
- Example

\[
h(n) = \begin{cases} 
0 & \text{if } n = 0, \\
n & \text{if } n > 4, \\
h(2 + h(2n)) & \text{if } n \leq 4.
\end{cases}
\]

- Famous example: Ackerman’s function.

\[
A(n, m) = \begin{cases} 
m + 1 & \text{if } n = 0, \\
A(n - 1, 1) & \text{if } n > 0, m = 0, \\
A(n - 1, A(n, m - 1)) & \text{otherwise.}
\end{cases}
\]

- \(A(3, m) = 2^{m+3} - 3\), \(A(4, m) = 2^{2^{\cdots^{2^{16}}}} - 3\), \(A(4, 1)\) exceeds the number of atoms in the universe.

- nice recursive expression, difficult iterative one.
Alternatives to (excessive) recursion

- Memoization: store previous results in a (hash) table. When function called recursively check first whether needed value is in the table.
- Of course, iterative solution. Need two previous values, so update two variables.

```c
unsigned int iterativeFib(unsigned int n) {
    if (n<2)
        return 1;
    else{
        register int i=2, tmp, current = 1, last =0;
        for(;i<=n;++i){
            tmp = current;
            current+=last;
            last=tmp;
        }
        last=tmp;
    }
    return current;
}
```
Recursion: concluding remarks

- Should be used with good judgement. No general rules when (not) to use it.
- Recursion usually less efficient than its iterative equivalent. But: if recursion 100 ms and iterative version 10ms, difference hardly perceivable.
- Recursion often simpler than its iterative equivalent and more consistent with logic of original algorithm.
- If nontail recursion, a stack has to be used.
- Two situations in which a nonrecursive implementation preferred.
- Real-time systems. Systems where an immediate response time vital for proper functioning of the program.
- Programs that are executed hundreds of times. E.g.: compiler.
- Avoid duplicating calls.
void BST::iterativePreorder() {
    Stack<BSTNode *> travStack;
    BSTNode *p = root;
    if (p != 0) {
        travStack.push(p);
        while (!travStack.empty()) {
            p = travStack.pop();
            visit(p);
            if (p->right != 0) {
                travStack.push(p->right);
            }
            if (p->left != 0) {
                travStack.push(p->left);
            }
        }
    }
}
Nonrecursive postorder tree traversal

- Recursive preorder and postorder only differ by order of operations.
- Can we easily transform iterative preorder into iterative postorder? **NO.**
- `iterativePreorder()`: visiting before both children pushed to the stack.
- Children pushed first, then node visited: **still preorder traversal**.
- What matters: `visit()` has to follow `pop()`, the latter precedes both calls of `push()`.
- Preorder: want to visit left child first so **push right child first**. **STACK**: last in first out.
Nonrecursive postorder tree traversal

- Sequence generated by left-to-right postorder traversal is the same as the reversed sequence generated by right-to-left preorder traversal (VRL order).
- Can use two stacks: one to visit each node in the reverse order after right-to-left preorder traversal finished.
- However: can develop function for postorder traversal that pushes onto stack a node that has two descendants, once before traversing its left subtree, once before traversing right subtree.
- Auxiliary pointer $q$ is used to distinguish between these two cases.
- Nodes with one descendant pushed only once, leaves don’t need to be pushed at all.
void BST::iterativePostorder()
{
  Stack<BSTNode *> travStack;
  BSTNode *p = root; *q = root;
  while(p != 0){
    travStack.push(p);
    while(!travStack.empty()){
      for( ;p->left != 0; p=p->left)// work in left subtree
        travStack.push(p);
      while(p!=0 && (p->right==0 || p->right == q)){
        visit(p); // right child: none or last visited node
        q=p; // q is last visited node
        if(travStack.empty()) return;
        p = travStack.pop();
      }
    }
    travStack.push(p);
    p = p->right; // work in right subtree
  }
}
Nonrecursive inorder. Stackless DF traversal

- Nonrecursive inorder: Very difficult. Only justified when speed is really paramount.
- Can eliminate use of stack if we use threaded trees.
- Threaded trees: stack is "part of the tree". Pointers to predecessor and successor of a node according to an inorder traversal.
- Alternative: overload pointer meaning. Left pointer: pointer to child or predecessor.
- Need new data member to indicate current meaning of the pointers.
- One thread may be sufficient.
Threaded trees

- Threaded trees: stack is "part of the tree". Pointers to predecessor and successor of a node according to an inorder traversal.
- Alternative: overload pointer meaning. Left pointer: pointer to child or predecessor.
- Need new data member to indicate current meaning of the pointers.
- One thread may be sufficient.
class ThreadedNode{
public:
    ThreadedNode(){
        left = right = 0;
    }
    ThreadedNode(int el,ThreadedNode *l=0,ThreadedNode *r=0){
        key = el; left = l; right = l; successor = 0;
    }
    int key;
    ThreadedNode *left,*right;
    unsigned int successor : 1;
}

class ThreadedTree{
public:
    ThreadedTree(){
        root = 0;
    }
}
Class ThreadedTree

```c++
void insert(int);
void inorder();

......

protected:
    ThreadedNode *root;
};

void ThreadedTree::inorder(){
    ThreadedNode *prev,*p=root;
    if (p!=0){ // process only nonempty trees;
        while(p->left != 0) // start at leftmost node
            p = p->left;
        while(p!=0){
            visit(p); prev = p; // prev= last visited node
            p = p->right; // after visiting go to the right
                // or successor node
                if (p != 0 && prev->successor == 0) //if descendent
                    while(p->left != 0) // go to the
                    p = p->left; // leftmost node
                // otherwise will visit the successor next time;
        }
    }
```
Threaded Trees: Preorder (idea)

- Can be used also for preorder and postorder traversals.
- Preorder: current node is visited first and then traversal continues with its left descendant, if any, or right descendant, if any.
- If current node is a leaf, threads are used to go through the chain of already visited inorder successors to restart traversal with the right descendant of the last successor.
Threaded Trees: Postorder (idea)

- Postorder: a dummy node created that has root as left descendant.
- A variable can be used to check type of current action.
- If action is left traversal and current node has a left descendant, then descendant is traversed. Otherwise action changed to right traversal.
- If action is right traversal and current node has a left descendant, action changed to left traversal. Otherwise action changed to visiting a node.
- If action is visiting node: current node is visited, afterwards its postorder successor has to be found.
- If current node's parent accessible through a thread (i.e. current node is parent's left child) then traversal is set to continue with the right descendant of parent.
- If current node has no right descendant, this is the end of the right-extended chain of nodes.
- First: the beginning of the chain is reached through the thread of the current node.
- Second: right references of nodes in the chain is reversed.
- Finally: chain is scanned backward, each node is visited, then right references are restored to previous settings.
Traversal through tree transformation

• Possible to traverse a tree without using any stack or threads by making temporary changes in trees during traversal.
• Changes: reassign some pointers.
• Tree might lose temporarily tree structure, needs to be restored before traversal finished.
• Algorithm, due to J. Morris, for inorder traversal.
• If tree has no left successors, inorder trivial.
• Temporarily transforms the tree so no left subtree. Has to keep information to restore it.
• Transformation: make current node the right child of the rightmost node in its left descendant.
• We retain the left pointer of the node moved down right subtree.
MorrisInorder()

while (not finished)
    if (node has no left descendant)
        visit it;
        go to the right;
    else
        make this node the right child of the rightmost node in its left descendant; // leaf !
        go to this left descendant;
Morris’s Algorithm

What happens next?
Notice the moved nodes retain their left pointers so the original shape can be regained
Morris inorder: implementation

```c
void BST::MorrisInorder()
{
    BSTNode *p = root, *tmp;
    while(p!=0)
        if (p->left == 0)
            {visit(p);
             p = p-> right;}
        else
            {
tmp = p->left; while(tmp->right != 0 &&
                // go to the rightmost node
                tmp->right != p) // of the left subtree or
                tmp = tmp-> right; // to the temporary parent
        if (tmp->right == 0)
            // of p; if 'true'
            tmp->right = p; // rightmost node was
                p = p->left; // reached, make it a
        } // temporary parent of the current root
```
else { // current root, else a temporary
    visit(p); // parent has been found; visit node p
    tmp->right = 0; // and then cut right pointer of p = p->right; // current parent, whereby it
} // ceases to be a parent;
}
Morris’s algorithm: Efficiency

• Notice: time depends on the number of loops.
• Number of loops: depends on number of left pointers.
• Some trees more efficient than others.
• Experimentally: 5 to 10% savings on randomly generated tree, but great space improvement.
• Preorder (idea): move visit() from the inner else clause to the inner if clause. A node visited before transformation.
• Postorder (idea): first create dummy node whose left descendant tree being processed. Then perform inorder traversal. In the inner else clause, after finding temporary parent, nodes between p->left and p (excluded) processed in reversed order.
Insertion

- searching does not modify the tree.
- To insert a new node with key el, a tree node with a dead end has to be reached, new node attached to it.
- found using same procedure as searching: compare key of currently scanned node to el. If el less than the key try left child; otherwise try right child.
- If the child is empty, discontinue search and make the child point to a new node of key el.
void BST::insert(int el) {
    BSTNode *p = root, *prev=0;
    while (p != 0) {
        prev = p;
        if (el > p->key) {
            p = p->right;
        } else {
            p = p->left;
        }
    }
    if (root == 0) {
        root = new BSTNode(el);
    } else {
        if (prev->key < el) {
            prev->right = new BSTNode(el);
        } else {
            prev->left = new BSTNode(el);
        }
    }
}
Inserting in threaded tree

- stack traversal: does not change the tree, Morris: restores it after traversal.
- second method: preparatory actions (threads) needed before traversal.
- Threads can be created before traversal and removed each time it's finished. If traversal infrequent a viable option.
- What if this is not the case? Need algorithm to update threads when inserting.
- Update function: for inorder, only takes care of successors.
- Node with a right child: has its successor somewhere in the right subtree, does not need a thread.
- Why? Threads are for "climbing up the tree", not for going down.
- A node with no right child has its successor somewhere. Inherits successor from parent.
- If a node becomes a left node, its parent is successor.
void ThreadedTree::insert(int el) {
    ThreadedNode *p, *prev = 0, *newNode;
    newNode = new ThreadedNode(el);
    if (root == 0) {
        root = newNode;
        return;
    }
    p = root;
    while (p != 0) {
        prev = p;
        if (p->key > el)
            p = p->left;
        else if (p->successor == 0) // go to the right node only if
            p = p->right; // it is a descendant, not a successor;
        else break;
    }
if(prev ->key > el){ // if newNode is left child of
  prev->left = newNode; // its parent, the parent
  newNode->successor = 1; // also becomes its successor
  newNode->right = prev;
}
else if (prev->successor == 1){ // if the parent of newNode
  newNode->successor = 1; // is not the rightmost node,
  prev->successor = 0; // make parent’s successor
  newNode->right = prev->right; // newNode’s successor
  prev->right = newNode;
}
else prev->right = newNode; // otherwise it has no successor
Deleting nodes

- Level of complexity of deletion depends on the position of the node in the tree. Three cases:
  - The node is a leaf: it has no children. Set appropriate pointer of parent to null, dispose of node.
  - The node has one child: Parent’s pointer is reset to point to the node’s child. This way nodes’s children are lifted up one level. Then node is disposed of.
  - Node has two children: No one step operation can be made, because parent’s pointer cannot point to both children at the same time.
- More than one solution.
- Deletion by merging: Make one tree out of left and right subtree and then attach to parent.
Deletion by merging: Idea

- How can one merge the trees? By tree property every key in left subtree smaller than every key in the right subtree.
- SOLUTION: Find in the left subtree the node with the largest key and make it a parent of the right subtree.
- Symmetrically: can find node with smallest key in the right subtree and make it a parent of left subtree.
- Desired node: rightmost node of left subtree.
- To locate it: move along this subtree, take right pointers until null encountered.
- This means the node has no right child, no danger of violating the BST property by merging trees.
Deletion by merging: implementation

```cpp
void BST::deleteByMerging(BSTNode * & node)
{
BSTNode * tmp = node;
if (node != 0) {
  if (!node->right) // node has no right child: its left
    node = node->left; // child (if any) is attached to its parent
  else if (node->left == 0) // node has no left child: its right
    node = node->right; // child is attached to its parent;
  else { // have to merge subtrees
    tmp = node->left; // 1. move left
    while (tmp->right != 0) // 2. and then right as far as possible
      tmp = tmp->right;
    tmp->right = node->right; // establish link between the
    // rightmost node of the left subtree and
    // the right subtree
    tmp = node; // 4.
    node = node->left; // 5.
  }
  delete tmp; // 6.
}
```
void BST::findAndDeleteByMerging(int el) {
    BSTNode *node = root, *prev = 0;
    while (node != 0) {
        if (node->key == el)
            break;
        prev = node;
        if (node->key < el)
            node = node->right;
        else node = node->left;
    }
    if (node != 0 && node->key == el) {
        if (node == root)
            deleteByMerging(root);
        else if (prev->left == node)
            deleteByMerging(prev->left);
        else deleteByMerging(prev->right);
    } else if (root != 0)
        cout << "key " << el << " is not in the tree." << endl;
    else cout << "the tree is empty" << endl;
}