Acknowledgments

• The course follows the book “Introduction to Algorithms”, by Cormen, Leiserson, Rivest and Stein, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.

• These slides are based on those developed by Michael Böhlen for this course.

   (See http://www.inf.unibz.it/dis/teaching/DSA/)

• The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course.

   (See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ/)
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
DSA, Chapter 6: Overview

- **Binary Search Trees**
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- **Red-Black Trees**
  - Properties
  - Rotations
  - Insertion
  - Deletion
Dictionaries

A *dictionary* $D$ is a dynamic data structure containing elements with a *key* and a *data* field.

A dictionary allows the operations:

- `search(D, k)`
  - *returns (a pointer to) an element* $x$
  - *such that* $x.key = k$
  - *(and returns null otherwise)*

- `insert(D, x)`
  - *adds the element (pointed to by)* $x$ *to* $D$

- `delete(D, x)`
  - *removes the element (pointed to by)* $x$ *from* $D$
Ordered Dictionaries

A dictionary D may have keys that are comparable (ordered domain)

In addition to the standard dictionary operations, we want to support the operations:

- $\text{min}(D)$
- $\text{max}(D)$

and

- $\text{predecessor}(D, x)$
- $\text{successor}(D, x)$
A List-based Implementation

Unordered list

- search, min, max, predecessor, successor: $O(n)$
- insert, delete: $O(1)$

Ordered list

- search, insert: $O(n)$
- min, max, predecessor, successor, delete: $O(1)$

What kind of list is needed to allow for $O(1)$ deletions?
Refresher: Binary Search

- Narrow down the search range in stages
  - `findElement(22)`
Run Time of Binary Search

• The range of candidate items to be searched is halved after comparing the key with the middle element
  ➔ binary search on arrays runs in $O(\log n)$ time
• What about insertion and deletion?
  – search: $O(\log n)$
  – min, max, predecessor, successor: $O(1)$
  – insert, delete: $O(n)$
• Challenge: implement insert and delete in $O(\log n)$
• Idea: extended binary search to dynamic data structures
  ➔ binary trees
Binary Trees (Java)

```java
class Tree {
    Node root;
}

class Node {
    int key;
    Data data;
    Node left;
    Node right;
    Node parent;
}
```

In what follows we ignore the info field of nodes
Binary Search Trees

A binary search tree (BST) is a binary tree $T$ with the following properties:

- each internal node stores an item $(k,d)$ of a dictionary
- keys stored at nodes in the left subtree of $x$ are less than or equal to $k$
- keys stored at nodes in the right subtree of $x$ are greater than or equal to $k$

Example BSTs for 2, 3, 5, 5, 7, 8
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
Tree Walks

Keys in a BST can be printed using “tree walks”

Option 1: Print the keys of each node between the keys in the left and right subtree

→ inorder tree traversal

```c
inorderTreeWalk(Node x)
    if x != NULL then
        inorderTreeWalk(x.left)
        print n.key
        inorderTreeWalk(x.right)
```
Tree Walks/2

- inorderTreeWalk is a divide-and-conquer algorithm
- It prints all elements in monotonically increasing order
- Running time $\Theta(n)$
inorderTreeWalk can be thought of as a projection of the BST nodes onto a one-dimensional interval.
Other Forms of Tree Walk

A preorder tree walk processes each node before processing its children.

```
preorderTreeWalk(Node x)
    if x != NULL then
        print x.key
        preorderTreeWalk(x.left)
        preorderTreeWalk(x.right)
```
Other Forms of Tree Walk/2

A postorder tree walk processes each node after processing its children.

```
postorderTreeWalk(Node x)
  if x ≠ NULL then
    postorderTreeWalk(x.left)
    postorderTreeWalk(x.right)
    print x.key
```
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion
- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
Searching a BST

To find an element with key $k$ in the tree rooted at node $n$
  – compare $k$ with $n.key$
  – if $k < n.key$, search for $k$ in $n.left$
  – otherwise, search for $k$ in $n.right$
Pseudocode for BST Search

Recursive version: divide-and-conquer

```
Node search(Tree t, int k)
    return nodeSearch(t.root,k)

Node nodeSearch(Node n, int k)
    if n = NULL then return NULL
    if k = n.key then return n
    if k < n.key
        then return nodeSearch(n.left,k)
    else return nodeSearch(n.right,k)
```
Pseudocode for BST Search

Iterative version

Node search(Tree t, int k)
    return nodeSearch(t.root,k)

Node nodeSearch(Node x, int k)
    p := x
    while p ≠ NULL and k ≠ p.key do
        if k < p.key
            then p := p.left
        else p := p.right
    return p

What is the loop invariant here?
Search Examples

• search(x, 11)
Search Examples/2

- Search\((x, 6)\)
Analysis of Search

• Running time on a tree of height $h$ is $O(h)$
• After the insertion of $n$ keys, the worst-case running time of searching is $O(n)$
BST Minimum (Maximum)

Find the node with the minimum key in the tree rooted at node $x$

- That is, the leftmost node in the tree, which can be found by walking down along the left child axis as long as possible

```plaintext
minNode(Node x)
while x.left ≠ NULL do
  x := x.left
return x
```

- Maximum: walk down the right child axis, instead
- Running time is $O(h)$, i.e., proportional to the height of the tree.
Successor

Given node $x$, find the node with the smallest key greater than $x.key$

- We distinguish two cases, depending on the right subtree of $x$

  - **Case 1:** The right subtree of $x$ is non-empty ($\text{succ}(x)$ inserted after $x$)
    - successor is the minimal node in the right subtree
    - found by returning $\text{minNode}(x.right)$
Successor/2

• **Case 2:** the right subtree of x is empty
  (succ(x), if any, was inserted before x)
  – The successor (if any) is the lowest ancestor of x
    whose left subtree contains x
  – Can be found by tracing parent pointers until the
    current node is the left child of its parent:
    return the parent
Successor Pseudocode

```
successor(Node x)
    if x.right ≠ NULL
        then return minNode(x.right)
    y := x
    while y.parent ≠ NULL and y = y.parent.right
        y = y.parent.
    y := y.parent
    return y.parent
```

For a tree of height \( h \), the running time is \( O(h) \)

*Note: no comparison among keys needed, since we have parent pointers!*
Successor with Trailing Pointer

Idea: Introduce yp to avoid derefencing y.parent

```
successor(Node x)
    if x.right ≠ NULL
        then return minNode(x.right)
    y := x
    yp := y.parent
    while yp ≠ NULL and y = yp.right do
        y := yp
    return yp
```
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
BST Insertion

The basic idea derives from searching:

- **construct** an element \( p \)
  whose left and right children are NULL
  and insert it into \( T \)

- **find the location** in \( T \)
  where \( p \) belongs to
  (as if searching for \( p\.key \)),

- **add** \( p \) there

The running time on a tree of height \( h \) is \( O(h) \)
Notice: Code uses technique of the trailing (= “one step delayed”) pointer

```plaintext
treeInsert(Tree t, Node n)
    front:=t.root; rear:=NULL;
    while front ≠ NULL do
        rear:=front;
        if n.key < front.key
            then front:=front.left
        else front:=front.right
        if rear = NULL //empty tree
            then t.root:=n;
                n.parent:=NULL;
        elsif n.key < rear.key
            then rear.left:=n;
        else rear.right:=n;
        n.parent:=rear;
```
BST Insertion Code (Java)

Realizes the pseudocode as a method of the class Tree

```java
void insert(Node n) { //insert n into current tree
    Node front = root; Node rear = NULL;
    while (front != NULL) {
        rear = front;
        if (n.key < front.key)
            front = front.left;
        else
            front = front.right;
    }
    if (rear == NULL) {// the tree is empty
        root = n;
        n.parent = null;
    } else if (n.key < rear.key)
        rear.left = n;
    else
        rear.right = n;
    n.parent = rear;
}
```
BST Insertion Example

Insert 8

```
3
 /   \
2   5
 /     \
4   8
```

```
5
 /   \
3   7
 /   /\  \
2   5 8 11
```
BST Insertion: Worst Case

In which order must the insertions be made to produce a BST of height $n$?
BST Sorting/2

Sort an array $A$ of $n$ elements using treeInsert and a version of inorderTreeWalk that inserts node keys into an array (instead of printing them)

```plaintext

treeSort(A)
    T := new Tree() // a new empty tree
    for i := 1 to A.length do
        treeInsert(T, new Node(A[i]))
        inorderTreeWalkPrintToArray(T,A)
```

We assume constructors

- `Tree()` produces empty tree
- `Node(int k)` produces a node with key $k$
BST Sorting/2

Sort the numbers

5 10 7 1 3 1 8

• Build a binary search tree

• Call inorderTreeWalk

1 1 3 5 7 8 10
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
Deletion

Delete node $x$ from a tree $T$

We distinguish three cases
\begin{itemize}
  \item $x$ has no child
  \item $x$ has one child
  \item $x$ has two children
\end{itemize}
Deletion Case 1

If $x$ has no children: simply remove $x$
Deletion Case 2

If $x$ has exactly one child, make the parent of $x$ point to that child and delete $x$
Deletion Case 3

• If x has two children:
  – find the largest child y in the left subtree of x (i.e., y is predecessor(x))
  – recursively remove y (note that y has at most one child), and
  – replace x with y.

• “Mirror” version with successor(x) [CLRS]
 BST Deletion Pseudocode

\begin{verbatim}
define delete(Tree t, Node x)
    if x.left = NULL or x.right = NULL
        drop := x
    else drop := successor(x)
    if drop.left ≠ NULL
        keep := drop.left
    else keep := drop.right
    if keep ≠ NULL
        keep.parent := drop.parent
    if drop.parent = NULL
        t.root := keep
    else if drop = drop.parent.left
        drop.parent.left := keep
    else drop.parent.right := keep
    if drop ≠ x
        x.key := drop.key
        // x.data := drop.data
\end{verbatim}
BST Deletion Code (Java)

- Java method for class Tree
- Version without “parent” field
- Note again the trailing pointer technique

```java
void delete(Node x) {
    front = root; rear = NULL;
    while (front != x) {
        rear := front;
        if (x.key < front.key)
            front := front.left;
        else
            front := front.right;
    } // rear points to a parent of x (if any)
    ...
}
```
BST Deletion Code (Java)/2

• x has less than 2 children
• fix pointer of parent of x

```java
... if (x.right == NULL) {
    if (rear == NULL) root = x.left;
    else if (rear.left == x) rear.left = x.left;
    else rear.right = x.left;
} else if (x.left == NULL) {
    if (rear == NULL) root = x.right;
    else if (rear.left == x) rear.left = x.right;
    else rear.right = x.right;
} else {
    ...
}```
BST Deletion Code (Java)/3

• x has 2 children

```java
succ = x.right; srear = succ;
while (succ.left != NULL)
    { srear:=succ; succ:=succ.left; }
if (rear == NULL) root = succ;
else if (rear.left == x) rear.left = succ;
else rear.right = succ;
succ.left = x.left;
if (srear != succ) {
    srear.left = succ.right;
    succ.right = x.right;
}
```
Balanced Binary Search Trees

- Problem: execution time for tree operations is $\Theta(h)$, which in worst case is $\Theta(n)$
- Solution: balanced search trees *guarantee* small height $h = O(\log n)$
Suggested Exercises

Implement a class of binary search trees with the following methods:

• max, min, successor, predecessor, search (iterative & recursive), insert, delete (swap with successor and predecessor), print, print in reverse order

• treeSort
Suggested Exercises/2

Using paper & pencil:

• Draw the trees after each of the following operations, starting from an empty tree:
  – insert 9, 5, 3, 7, 2, 4, 6, 8, 13, 11, 15, 10, 12, 16, 14
  – delete 16, 15, 5, 7, 9
     (both with successor and predecessor strategies)

• Simulate the following operations after the above:
  – Find the max and minimum
  – Find the successor of 9, 8, 6
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
Java's TreeMap

```java
public class TreeMap<K,V> extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
Red/Black Trees

A red-black tree is a binary search tree with the following properties:

1. Nodes are colored red or black
2. NULL leaves are black
3. The root is black
4. No two consecutive red nodes on any root-leaf path
5. Same number of black nodes on any root-leaf path (called black height of the tree)
RB-Tree Properties

Some measures
- $n$ – # of internal nodes
- $h$ – height
- $bh$ – black height

- $2^{bh} - 1 \leq n$
- $h/2 \leq bh$
- $2^{h/2} \leq n + 1$
- $h \leq 2 \log(n + 1)$

$\Rightarrow$ balanced!
RB-Tree Properties/2

• Operations on a binary-search tree (search, insert, delete, ...) can be accomplished in $O(h)$ time

• The RB-tree is a binary search tree, whose height is bounded by $2 \log(n + 1)$, thus the operations run in $O(\log n)$

  Provided that we can maintain the red-black tree properties spending no more than $O(h)$ time on each insertion or deletion
DSA, Chapter 6: Overview

– Binary Search Trees
  • Tree traversals
  • Searching
  • Insertion
  • Deletion

– Red-Black Trees
  • Properties
  • Rotations
  • Insertion
  • Deletion
Rotation

Right rotation of B

Left rotation of A
Right Rotation

\textbf{RightRotate}(Node B)
\begin{align*}
  A & := B.\text{left} \\
  B.\text{left} & := A.\text{right} \\
  B.\text{left}.\text{parent} & := B
\end{align*}

\textbf{if} (B = B.\text{parent}.\text{left}) \textbf{then} B.\text{parent}.\text{left} := A
\textbf{if} (B = B.\text{parent}.\text{right}) \textbf{then} B.\text{parent}.\text{right} := A
A.\text{parent} := B.\text{parent}

A.\text{right} := B
B.\text{parent} := A
The Effect of a Rotation

• **Maintains inorder key ordering**
  For all $a \in \alpha$, $b \in \beta$, $c \in \gamma$
  rotation maintains the invariant (for the keys)
  $a \leq A \leq b \leq B \leq c$

• After right rotation
  – depth($\alpha$) decreases by 1
  – depth($\beta$) stays the same
  – depth($\gamma$) increases by 1

• Left rotation: symmetric

• Rotation takes $O(1)$ time
DSA, Chapter 6: Overview

– Binary Search Trees
  • Tree traversals
  • Searching
  • Insertion
  • Deletion

– Red-Black Trees
  • Properties
  • Rotations
  • Insertion
  • Deletion
**Insertion in the RB-Trees**

\[
\text{rBInsert}(\text{RBTree } t, \text{ RBNode } n) \\
\quad \text{Insert } n \text{ into } t \text{ using} \\
\quad \text{the binary search tree insertion procedure} \\
\quad n.\text{left} := \text{NULL} \\
\quad n.\text{right} := \text{NULL} \\
\quad n.\text{color} := \text{red} \\
\quad \text{rBInsertFixup}(n)
\]
Fixing Up a Node: Intuition

Case 0: parent is black
   ➞ ok

Case 1: both parent and uncle are red
   ➞ change colour of parent/uncle to black
   ➞ change colour of grandparent to red
   ➞ fix up the grandparent

Exception: grandparent is root ➞ then keep it black

Case 2: parent is red and uncle is black, and
   node and parent are in a straight line
   ➞ rotate at grandparent

Case 3: parent is red and uncle is black, and
   node and parent are not in a straight line
   ➞ rotate at parent  (leads to Case 2)
Insertion

Let

\[ n = \text{the new node} \]
\[ p = n.\text{parent} \]
\[ g = p.\text{parent} \]

In the following assume

\[ p = g.\text{left} \]
Insertion: Case 0

Case 0: \( p.color = \text{black} \)

- No properties of the tree are violated
- We are done
Insertion: Case 1

Case 1: n’s uncle $u$ is red

- Action
  
  $p$.color := black
  $u$.color := black
  $g$.color := red
  $n$ := $g$

- Note: the tree rooted at $g$ is balanced enough (black depth of all descendants remains unchanged)
Insertion: Case 2

Case 2: n’s uncle u is black and n is a left child

- Action
  
  p.color := black
  
  g.color := red
  
  RightRotate(g)

- Note: the tree rooted at g is balanced enough (black depth of all descendents remains unchanged).
## Insertion: Case 3

### Case 3: n’s uncle u is black
and n is a right child

- **Action**
  
  \[
  \text{LeftRotate(p)}
  \]

  
  \[n := p\]

- **Note:** The result is a Case 2
Insertion: Mirror Cases

• All three cases are handled analogously if \( p \) is a right child

• Exchange left and right in all three cases
Insertion: Case 2 and 3 Mirrored

Case 2m: n’s uncle \( u \) is black and \( n \) is a right child

- Action

\[
\begin{align*}
    p\cdot \text{color} & := \text{black} \\
g\cdot \text{color} & := \text{red} \\
\text{Left} \text{Rotate}(g)
\end{align*}
\]

Case 3m: n’s uncle \( u \) is black and \( n \) is a left child

- Action

\[
\begin{align*}
    \text{Right} \text{Rotate}(p) \\
n & := p
\end{align*}
\]
Insertion Summary

- If two red nodes are adjacent, we perform either
  - a restructuring (with one or two rotations) and stop (cases 2 and 3), or
  - recursively propagate red upward (case 1)
- A restructuring takes constant time and is performed at most once; it reorganizes an off-balanced section of the tree
- Propagations may continue up the tree and are executed $O(\log n)$ times (height of the tree)
- The running time of an insertion is $O(\log n)$
An Insertion Example

Insert "REDSOX" into an empty tree

Now, let us insert "CUBS"
Insert C (Case 0)
Insert U (Case 3, Mirror)
Insert U/2
Insert B (Case 2)

```
    E
   / 
  D   R
 / \
C   O
     /
    U
     / \  \
    S   X
```

```
    E
   / 
  D   R
 / \
C   O
     /
    U
     / \  \
    B   X
```
Insert B/2
Insert S (Case 1)
Insert $S/2$ (Case 2 Mirror)
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
Deletion

We first apply binary search tree deletion

- We can easily delete a node with at least one NULL child
- If the key to be deleted is stored at a node u with two children, we replace its content with the content of the largest node v of the left subtree (the predecessor of u) and delete v instead
Deletion Algorithm

1. Remove \( u \)

2. If \( u\).color = red we are done;
   else, assume that \( v \) (the predecessor of \( u \))
   gets an additional black color:
   - if \( v\).color = red then \( v\).color = black
     and we are done!
   - else \( v \)’s color is “double black”
Deletion Algorithm/2

How to eliminate double black edges?

- The intuitive idea is to perform a color compensation
  Find a red node nearby, and change the pair (red, double black) into (black, black)

- Two cases: restructuring and recoloring
- Restructuring resolves the problem locally, while recoloring may propagate it upward.

Hereafter we assume ν is a left child (swap right and left otherwise)
Deletion Case 1

Case 1: \( v \)'s sibling \( s \) is \textbf{black}

and both children of \( s \) are \textbf{black}

- Action: recoloring

\[
\begin{align*}
 & s.\text{color} := \text{red} \\
 & v.\text{color} := \text{black} \\
 & p.\text{color} := p.\text{color} + \text{black}
\end{align*}
\]

- Note: We reduce the black depth of both subtrees of \( p \) by 1;
parent \( p \) becomes more black
Deletion: Case 1

If parent $p$ becomes **double black**, continue upward
Deletion: Case 2

Case 2: \( v \)'s sibling \( s \) is black
and \( s \)'s right child is red

- Action
  
  \[
  \begin{align*}
  s.\text{color} & = p.\text{color} \\
  p.\text{color} & = \text{black} \\
  s.\text{right}.\text{color} & = \text{black}
  \end{align*}
  \]
  LeftRotate(p)

- Idea: Compensate the extra black ring of \( v \) by the red of \( r \)
- Note: Terminates after restructuring
Deletion: Case 3

Case 3: v’s sibling s is black, s’s left child is red, and s’s right child is black

- Idea: Reduce to Case 2
- Action
  s.left.color = black
  s.color = red
  RightRotation(s)
  s = p.right

- Note: This is now Case 2
Deletion: Case 4

Case 4: \( v \)'s sibling \( s \) is red

- Idea: give \( v \) a black sibling
- Action
  
  \[
  \begin{align*}
  s.\text{color} &= \text{black} \\
  p.\text{color} &= \text{red} \\
  \text{LeftRotation}(p) \\
  s &= p.\text{right}
  \end{align*}
  \]

- Note: This is now a Case 1, 2, or 3
Delete 9
Delete 9/2

- Case 2 (sibling is black with black children) – recoloring
Delete 8

Before deletion:

```
  6
 /   \
4     8
 / \
2   5
```

After deletion:

```
  6
 /   \
4     \\
2     7
```
Delete 7: Restructuring
How Long Does it Take?

Deletion in a RB-tree takes $O(\log n)$

Maximum:

- three rotations and
- $O(\log n)$ recolorings
Suggested Exercises

• Add left-rotate and right-rotate to the implementation of your binary trees

• Implement a class of red-black search trees with the following methods:
  – (...), insert, delete,
Suggested Exercises/2

Using paper and pencil:

• Draw the RB-trees after each of the following operations, starting from an empty tree:
  1. Insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
  2. Delete 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

• Try insertions and deletions at random
Other Balanced Trees

- Red-Black trees are related to **2-3-4 trees** (non-binary)

- **AVL-trees** have simpler algorithms, but may perform a lot of rotations
Next Part

• Hashing