CMPT231

Lecture 5: ch10, 12

Dynamic Data Structures:

Linked Lists and Binary Search Trees

Psalm 104:14,16-17 (NASB) (p.1/2)

14 He causes the **grass** to grow for the cattle,
And **vegetation** for the labor of man,
So that he may bring forth **food** from the earth,

16 The **trees** of the Lord drink their fill,
The **cedars** of Lebanon which He planted,

17 Where the **birds** build their nests,
And the **stork**, whose home is the fir trees.

Psalm 104:27-28,30 (NASB) (p.2/2)

27 They all **wait** for You
To give them their **food** in due season.

28 You **give** to them, they **gather** it up;
You **open** Your hand, they are **satisfied** with good.

30 You send forth Your **Spirit**, they are **created**;
And You **renew** the face of the ground.

Outline for today

- Review of **pointers**
- **Linked lists**: singly/**doubly**-linked, **circular**
- **Stacks** and **queues**
- **Trees** and Binary search trees (**BST**)
  - BST **traversals**
  - **Searching**
  - **Min**/max and **successor**/predecessor
  - **Insert** and **delete**
  - **Randomised** BST
- **Skip lists**

Stack frame vs heap

- Where are program **variables** stored in memory?
  - **Static** vars + formal **parameters** ⇒ **stack frame**
    - Size known at **compile** time
  - **Local** vars dynamically allocated ⇒ **heap**
    - Typically **deallocated** when program/function exits
- A **pointer** is a var whose **value** is a memory **location** in the heap
  - Usually has its own **type** (e.g., "pointer to float")
  - But really is just an **unsigned int** itself!

Using pointers

- **Declare** a var of type "pointer to int" (or other type)
- Get **address** of a var by using "address-of" operator
- Read from **memory location** with "dereference" operator

```
int myAge = 20;
int* myAgePtr = &myAge;     // get address of myAge
cout << *myAgePtr;
  // dereference: prints "20"
```

Pointer **arithmetic** can be dangerous!

```
*( myAgePtr + 1 );
  // segmentation fault!
*( 1033813 );
       // random spot in memory!
```

Pointer-less languages

- To prevent **segfaults**, most languages (besides C/C++) do **not** have explicit pointers
- Instead, you create **references** (“aliases”)
  - Variables are simply entries in a **namespace**
  - Map an **identifier** to a **location** in the heap
  - **Multiple** identifiers can map to same location
- Be aware of when a **reference** is made vs a **copy**!

```
ages = [ 3, 5, 7, 9 ]       # Python list (mutable)
myAges = ages
       # create alias
myAges[ 2 ] = 11
    # overwrite 7
ages

# [ 3, 5, 11, 9 ]
```

Outline for today

- Review of pointers
- **Linked lists: singly/doubly-linked, circular**
- Stacks and queues
- Trees and Binary search trees (BST)
    - BST traversals
    - Searching
    - Min/max and successor/predecessor
    - Insert and delete
    - Randomised BST
- Skip lists

Linked lists

- Linear, **array-like** data structure, but:
- **Dynamic**: can change length (not fixed at compile time)
- Fast mid-list **insert** / delete (vs **shifting**)
- But **random** access slower than array

```
class Node:
  def __init__( self, key=None, next=None ):
    ( self.key, self.next ) = ( key, next )

head = Node( 9 )
head = Node( 7, head )
head = Node( 5, head )
head = Node( 3, head )
```

Doubly-linked lists

- Track **both** .prev and .next pointers in each node
- Allows us to move forwards and **backwards** through list

```
class Node:
  def __init__( self, key=None, prev=None, next=None ):
    self.key = key
    self.prev = prev
    self.next = next
```

`![doubly-linked list](static/img/Fig-10-3a.svg)`

Node vs LinkedList

- Create a wrapper **datatype** for the overall **list**
- Keep both **head** and **tail** pointers
  - So we can **jump** directly to either end of the list

```
class LinkedList:
  def __init__( self, head=None, tail=None ):
    ( self.head, self.tail ) = ( head, tail )

x = LinkedList( Node( 3 ), Node( 5 ) )
x.head.next = x.tail
x.tail.prev = x.head
```

What **result** does this code produce?

Circularly-linked lists

- .next pointer of **tail** node wraps back to **head**
- When **traversing** list, ensure not to **circle** forever!
  - e.g., store **length** of list,
    and **track** how many nodes we've traversed
  - or, add a **sentinel** node with special key
- Circularly-linked lists can also be **doubly**-linked
  - **Both** .prev and .next pointers wrap around

`![Circular doubly-linked list](static/img/Fig-10-4b.svg)`

Insert into linked list

- **Create** new node with the given key
  - and prepend to the **head** of the list
- Also called **push** since it only inserts at **head**

```
class LinkedList:
  def insert( self, key=None ):
    self.head = Node( key, self.head )
```

Try it: insert 3 into list `[5, 7, 9]`

Search on linked list

- Return a **reference** to a node with the given key
  - Or return None if key not found in list

```
class LinkedList:
  def search( self, key ):
    cur = self.head
    while cur != None:
      if cur.key == key:
return cur
      cur = cur.next
    return None
```

Try it: search for 7 in list `[3, 5, 7, 9]`

Delete from linked list

- **Splice** a referenced node out of the list
  - If given key instead of pointer, just **search** first
- **Update** .prev/.next links in neighbouring nodes
  - So they **skip** over the deleted node
- **Free** the unused memory so it can be reused
  - **Garbage**: allocated but unused/unreachable memory
  - **Memory leak**: heap grows indefinitely

```
class LinkedList:
  def delete( self, key ):
    node = self.search( key )
    node.prev.next = node.next      # what if deleting head/tail?
    node.next.prev = node.prev
    del node
```

Outline for today

- Review of pointers
- Linked lists: singly/doubly-linked, circular
- **Stacks and queues**
- Trees and Binary search trees (BST)
  - BST traversals
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete
  - Randomised BST
- Skip lists

Stacks and queues

- **Stack** ("LIFO"): last-in-first-out (memo spike)
- **Queue** ("FIFO"): first-in-first-out (pipeline)
- **Interface** (defining set of operations):
  - `length()`, `isempty()`: **number** of items
  - `push(item)`: **add** item to stack/queue
  - `peek()`: get item **without** deleting
  - `pop()`: peek and **delete** item
- **Underflow**: peek/pop on an **empty** stack/queue
- **Overflow**: push on a **full** stack/queue

Implementing stacks/queues

- Abstract data type (ADT): only specifies **interface**
- Can be **implemented** using arrays, linked-lists, or other
  - Memory usage, computational efficiency **trade-offs**

```
class Stack:
  def __init__( self ):
    self.head = None

  def push( self, key ):
    # overflow not a worry
    self.head = Node( key, self.head )

  def pop( self ):

  # watch for underflow!
    item = self.head
    key = item.key
```

Outline for today

- Review of pointers
- Linked lists: singly/doubly-linked, circular
- Stacks and queues
- **Trees and Binary search trees (BST)**
  - **BST traversals**
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete
  - Randomised BST
- Skip lists

Implementing trees

- For **Binary** trees, use 3 pointers:
  - **Parent**, **left** child, **right** child
- For **d-way** trees (unknown **degree** d):
  - **Parent**, **first** child, **next** sibling

`![binary tree](static/img/Fig-10-9.svg)` `![d-way tree](static/img/Fig-10-10.svg)`

Special trees for fast search

- Impose additional **constraints** on the tree
- Optimise for fast **search** and insert/delete:
  - Run in V (height of tree): if **full**, this is V (lg n)
- Can be used to implement a **dictionary** or **priority queue**
- Various **types**:
  - **Binary search** tree (BST) (ch12),
  - **Red-black** tree (ch13)
  - **B-tree** (ch18)
  - and many others!

Binary search trees (BST)

- **BST property**: at any node x in the tree,
  - $y \leq x$ $\forall$ nodes y in x's **left** sub-tree
  - $y \geq x$ $\forall$ nodes y in x's **right** sub-tree

`![BST with 6 nodes](static/img/bst-538.svg)`

Tree traversals

- **Touch** each node in tree
- **Preorder**: print **self** before children
  - *Example:* 532458

`![BST with 6 nodes](static/img/bst-538.svg)`

```
def preorder( node ):
  print node.key
  preorder( node.left )
  preorder( node.right )
```

- **Postorder**: print both **children** before self
  - Output? Pseudocode?
- **Inorder**: print **left** child, then **self**, then **right**
  - Output? Pseudocode?
- How to **list** elements of a **BST** in sequence?

Expression trees

- Trees are used for parsing and evaluating **expressions**:
  - e.g., tree for `( 2 * (-4) ) + 9`
  - Which **traversal** produces this expression?
  - **Try it**: draw tree for `2 * (-4 + 9)`
- Reverse Polish Notation (**RPN**):
  - e.g., `2, 4, -, *, 9, +`
  - Which **traversal** produces RPN?
- Can implement an RPN **calculator** using a **stack**
  - **Try it** on the above expression

Outline for today

- Review of pointers
- Linked lists: singly/doubly-linked, circular
- Stacks and queues
- Trees and Binary search trees (BST)
  - BST traversals
  - **Searching**
  - **Min/max and successor/predecessor**
  - Insert and delete
  - Randomised BST
- Skip lists

Searching a BST

- Compare with node's **key** to see which **subtree** to recurse
- **Complexity**: O(height of tree): if **full**, this is $O(\lg n)$
  - **Worst-case**: tree degenerates to **linked-list**!
  - Want to keep tree **balanced**

```
def search( node, key ):
  if (node == None) or (node.key == key):
    return node
  if key < node.key:
    return search( node.left, key )
  else:
    return search( node.right, key )
```

Min/max in BST

- `min()`: find the **smallest** key:
  - Keep taking **left** child as far as possible
- Similarly for `max()`
- **Iterative** solution faster than **recursive**:

```
def min( node ):
  while node.left != None:
    node = node.left
  return node.key
```

Successor / predecessor

- **Successor** of a node is **next** in line in an **in-order** traversal
  - **Predecessor**: previous in line
- If **right** subtree is **not** NULL:
  - Successor is **min** of right subtree
- Else, walk **up** tree until a parent link turns **right**:

```
def successor( node ):
  if node.right != NULL:
    return min( node.right )
  (cur, par) = (node, node.parent)
  while (par != NULL) and (cur == par.right
    (cur, par) = (par, par.parent)
  return par
```

![big BST]
(static/img/Fig-12-2.svg)

Try it: succ(*7*), pred(*6*), succ(*13*)

Outline for today

- Review of pointers
- Linked lists: singly/doubly-linked, circular
- Stacks and queues
- Trees and Binary search trees (BST)
    - BST traversals
    - Searching
    - Min/max and successor/predecessor
    - **Insert and delete**
    - **Randomised BST**
- Skip lists

Insert into BST

Search to find where to add node:

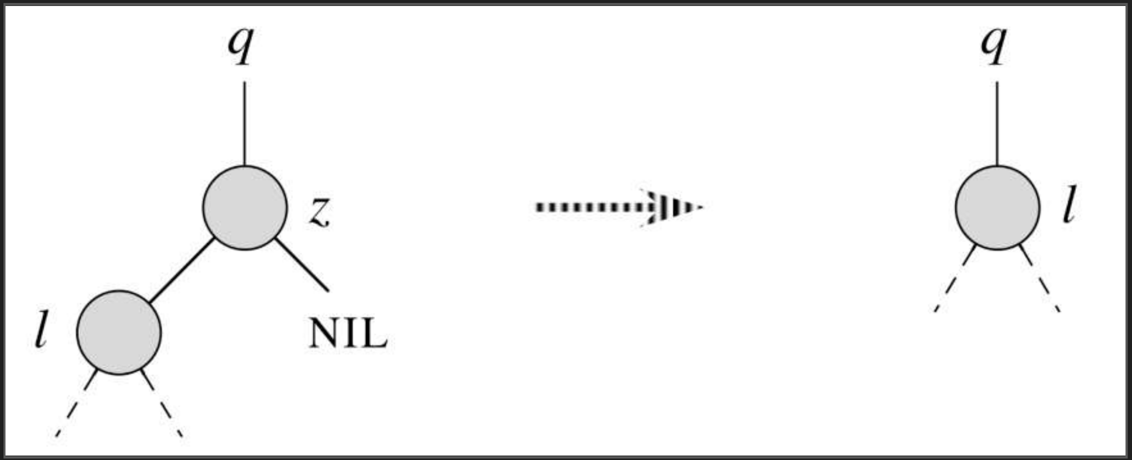
```
def insert( root, key ):
  cur = root
  while cur != NULL:
    if key < cur.key:
   # go left
      if cur.left == NULL:
cur.left = new Node( key )
cur.left.parent = cur
return
      cur = cur.left
    else:

<pre><code>   # go right
  if cur.right == NULL:</code></pre>
```

```
cur.right = new Node( key )
cur.right.parent = cur
return
```

`![big BST](static/img/Fig-12-2.svg)`

Delete from BST

- Deleting a **leaf** is easy (update links)
- If node z has **one** child l, **promote** it in place of z
  - Bring child's **subtrees** along
- If **two** children, replace it with its **successor**
  - We know successor is in **right** subtree, and has **no** left child (why?)

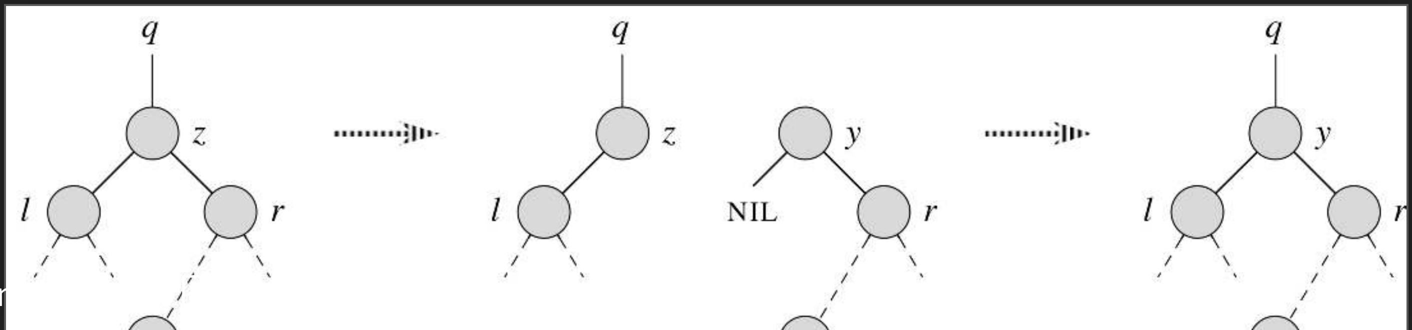


Delete node with two children

If successor *y* is the **right child** of the node *z* to be deleted, just **promote** it (*y* has no left child)

![12-4c, direct
successor]
(static/img/Fig-12-
4c.svg)

- Else, replace successor y with successor's own right child x
  - Replace node to be deleted z with successor y
  - z's old right subtree becomes y's right subtree



Randomly-built BST

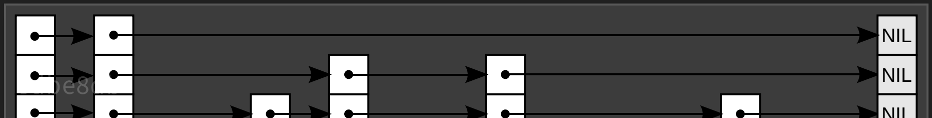
- How to **build** a BST from a set of n distinct keys?
- **Order** of insertion matters!
- Worst-case, BST becomes a **linked list**
  - **Search** (and hence insert, delete, etc.) is V (n)
- Try a (Fisher-Yates) **shuffle** of the keys first
  - Each of the n! **permutations** is equally likely
  - **Doesn't** mean each **BST** is equally likely (try it: n=3)
- Expected (average) **height** of random BST is V (lg n)
  - **Proof** in textbook

Outline for today

- Review of pointers
- Linked lists: singly/doubly-linked, circular
- Stacks and queues
- Trees and Binary search trees (BST)
  - BST traversals
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete
  - Randomised BST
- **Skip lists**

Skip lists

- **BST**-style searching applied to a **linked list** structure
  - Add **extra** links to nodes in linked list
- Each **level** only links p=1/2 of the nodes from the level **below** it, chosen **randomly**
- **First** node is special (-∞), linked in **every** level
- **Search** / insert / delete: work from top level downward
  - **Step** through list until key is **larger** than target
  - Go to **previous** node and step **down** a level, then **repeat**



Outline for today

- Review of **pointers**
- **Linked lists**: singly/**doubly**-linked, **circular**
- **Stacks** and **queues**
- **Trees** and Binary search trees (**BST**)
  - BST **traversals**
  - **Searching**
  - **Min**/max and **successor**/predecessor
  - **Insert** and **delete**
  - **Randomised** BST
- **Skip lists**