CS2020
Data Structures and Algorithms

Welcome!

Friday Recitations

Scheduling:

- CORS allocated 5 slots
- We are only doing 3 recitation sections

Constraints:

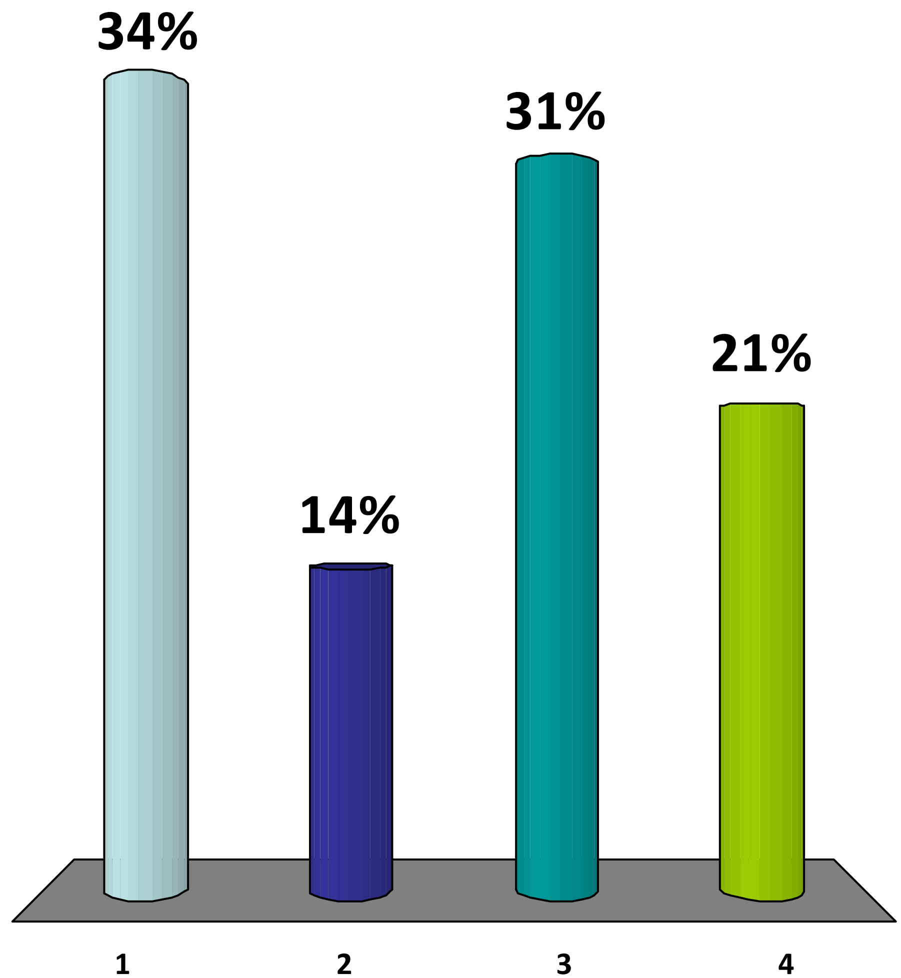
- Friday afternoon: 1-6pm

Options:

- 2pm, 3pm, 4pm
- 1pm, 2pm 4pm
- 1pm, 4pm, 5pm

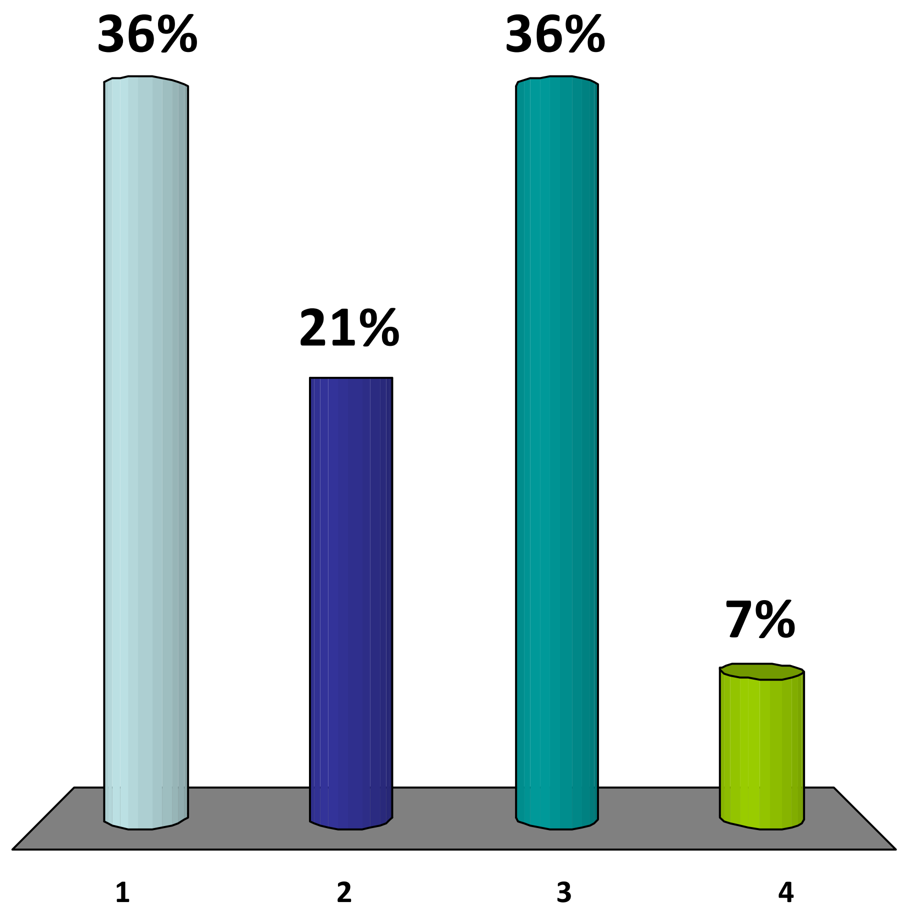
Option 1:

1. 2pm
2. 3pm
3. 4pm
4. NONE



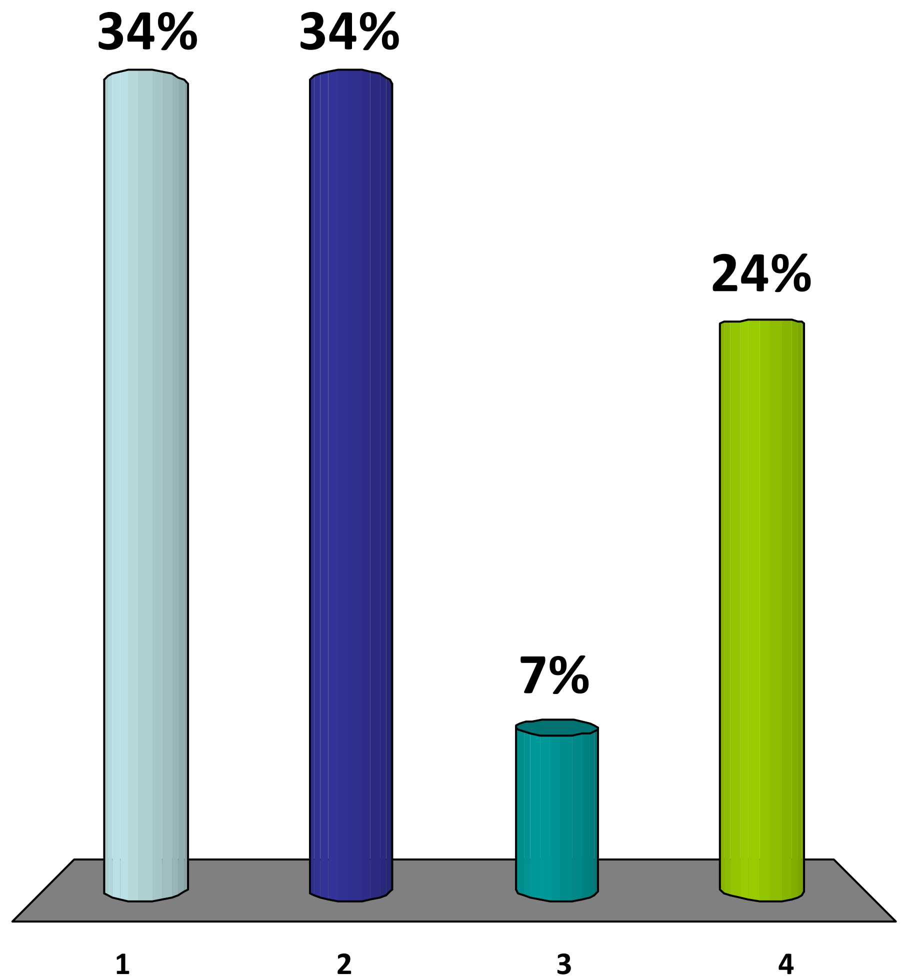
Option 2:

1. 1pm
2. 2pm
3. 4pm
4. NONE



Option 3:

1. 1pm
2. 4pm
3. 5pm
4. NONE



Problem Set 1

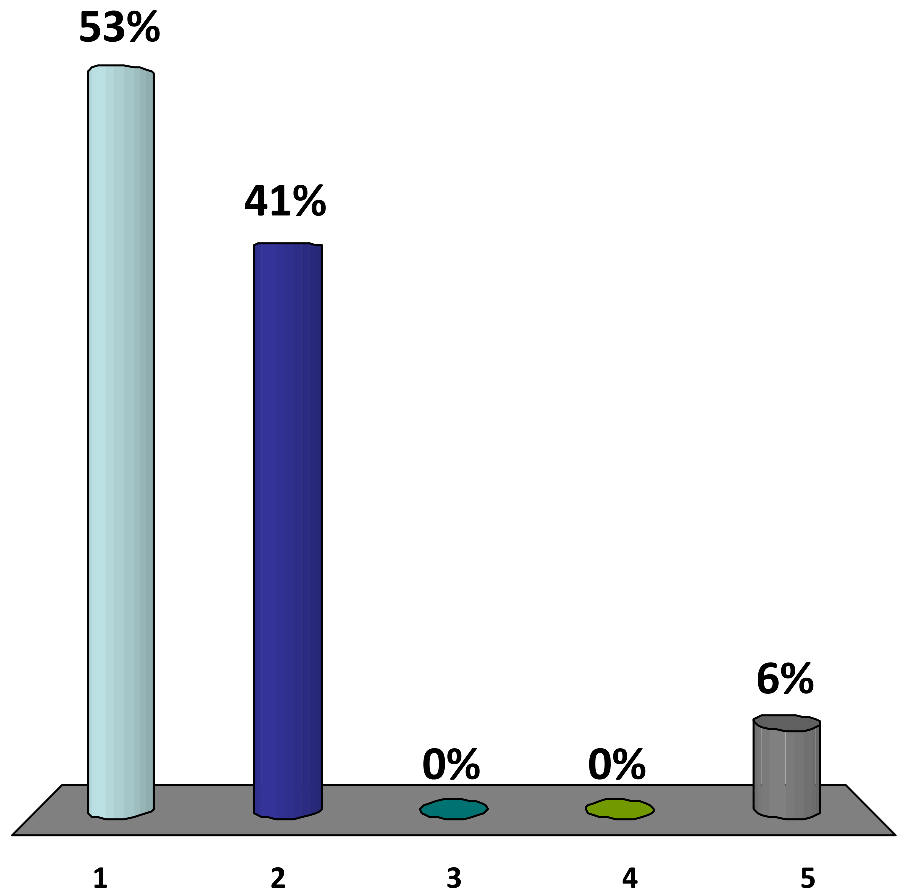
Due: Wednesday, 2pm

Issues:

- Eclipse
- TPTP
- Graphing

Problems with Eclipse?

1. No problems at all.
2. A few problems, at first, but ok.
3. Could not profile.
4. Could not write/compile Java.
5. Total FAIL.



Problem Set 2

Due: Wednesday, 2pm

- Incomplete?
- Improperly specified question?
- Typos?

Two solutions:

- Check with tutor/me.
- State (and justify) your assumptions.

Problem Set 3

To be released: Tuesday

- Do problem set 2 now!
- Start problem set 3 on Tuesday!
  - More programming…
  - More debugging…
  - More time…
- Advice:
  - Download each set of problems immediately.
  - Read the problems.

Today's Plan

Abstract data types

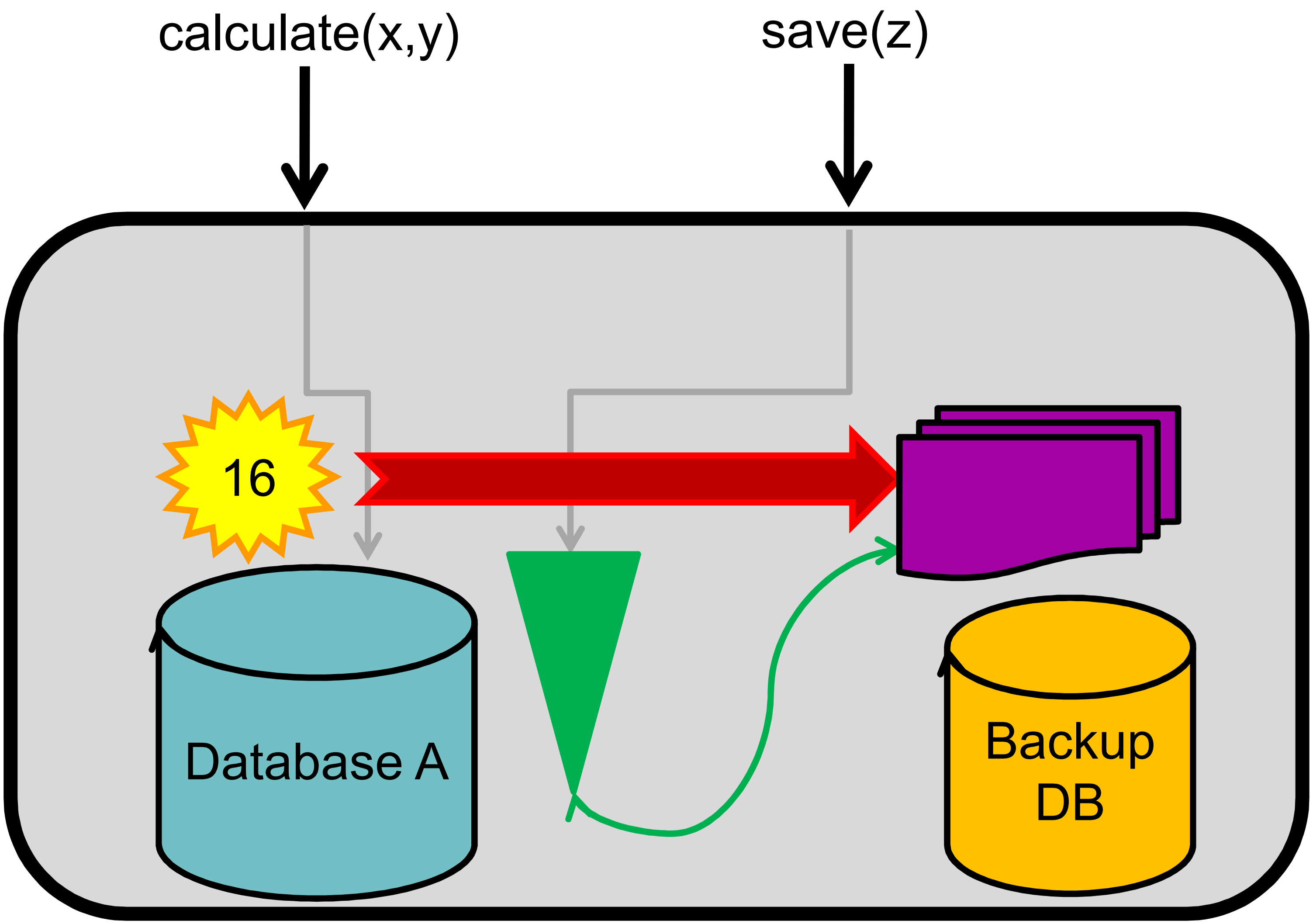
- Motivation
- Examples
- Java

Problem: Scheduling Airplanes

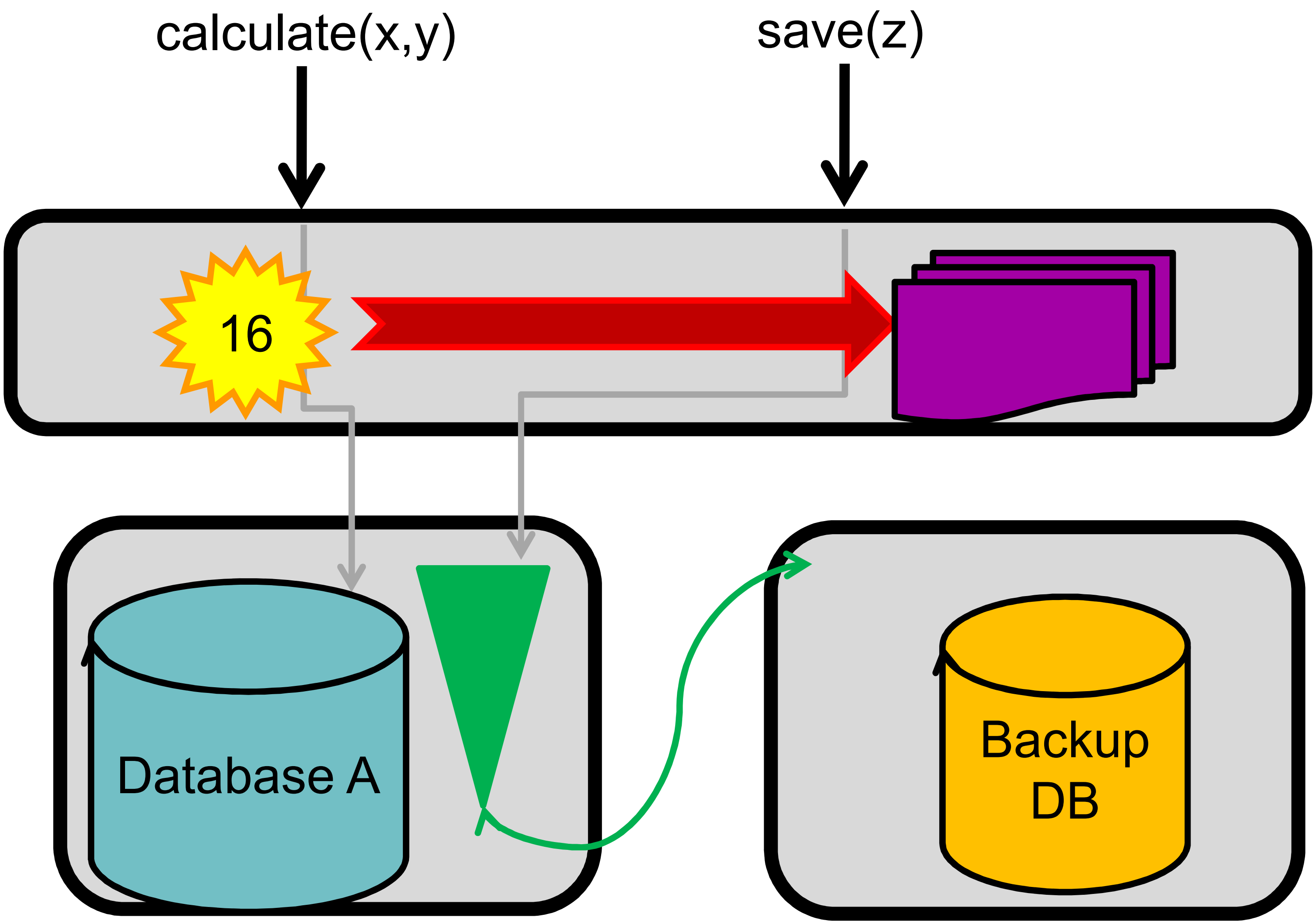
- Dynamic Dictionary
- Binary Search Trees
- Augmented trees

Abstraction

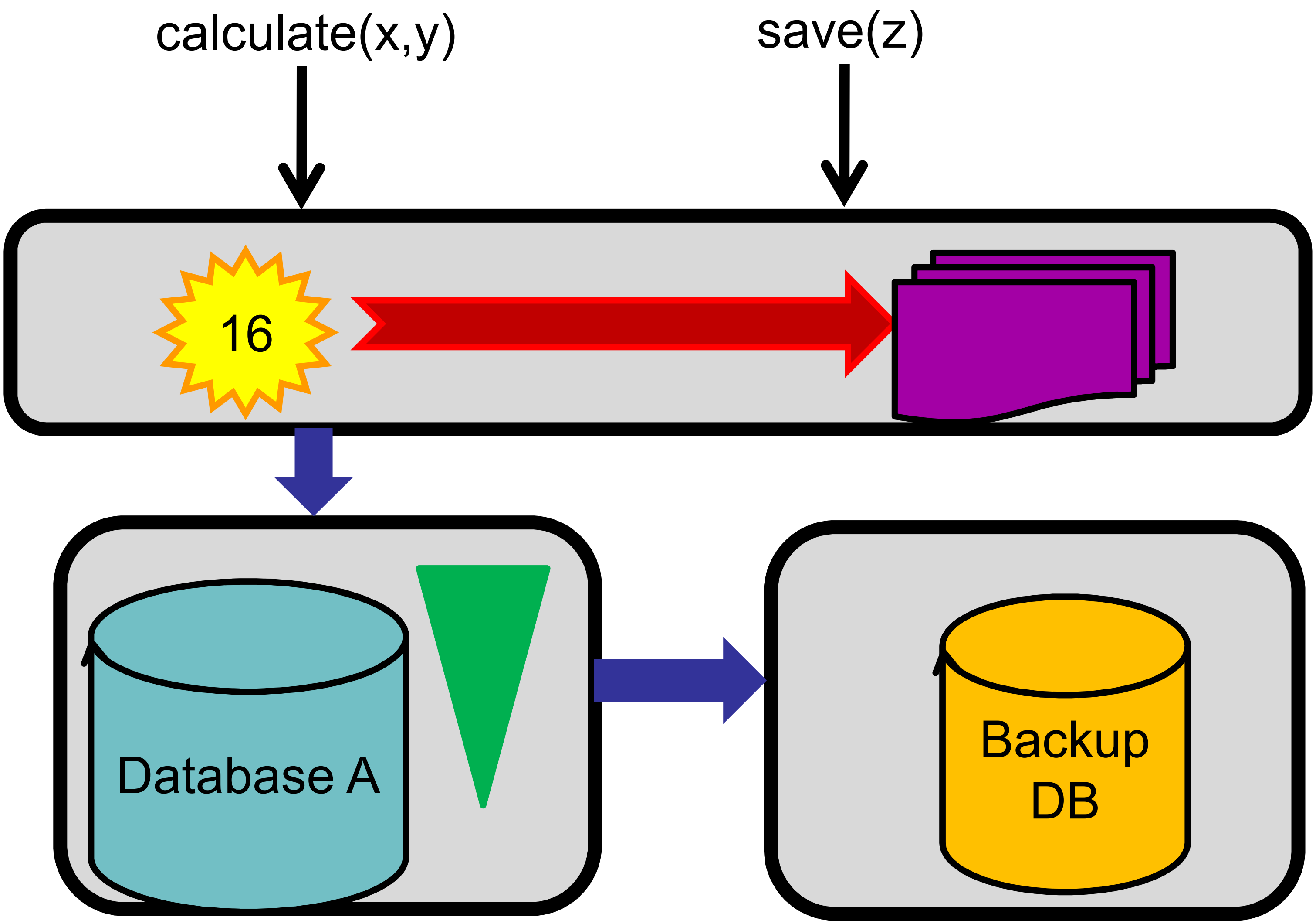
Abstraction



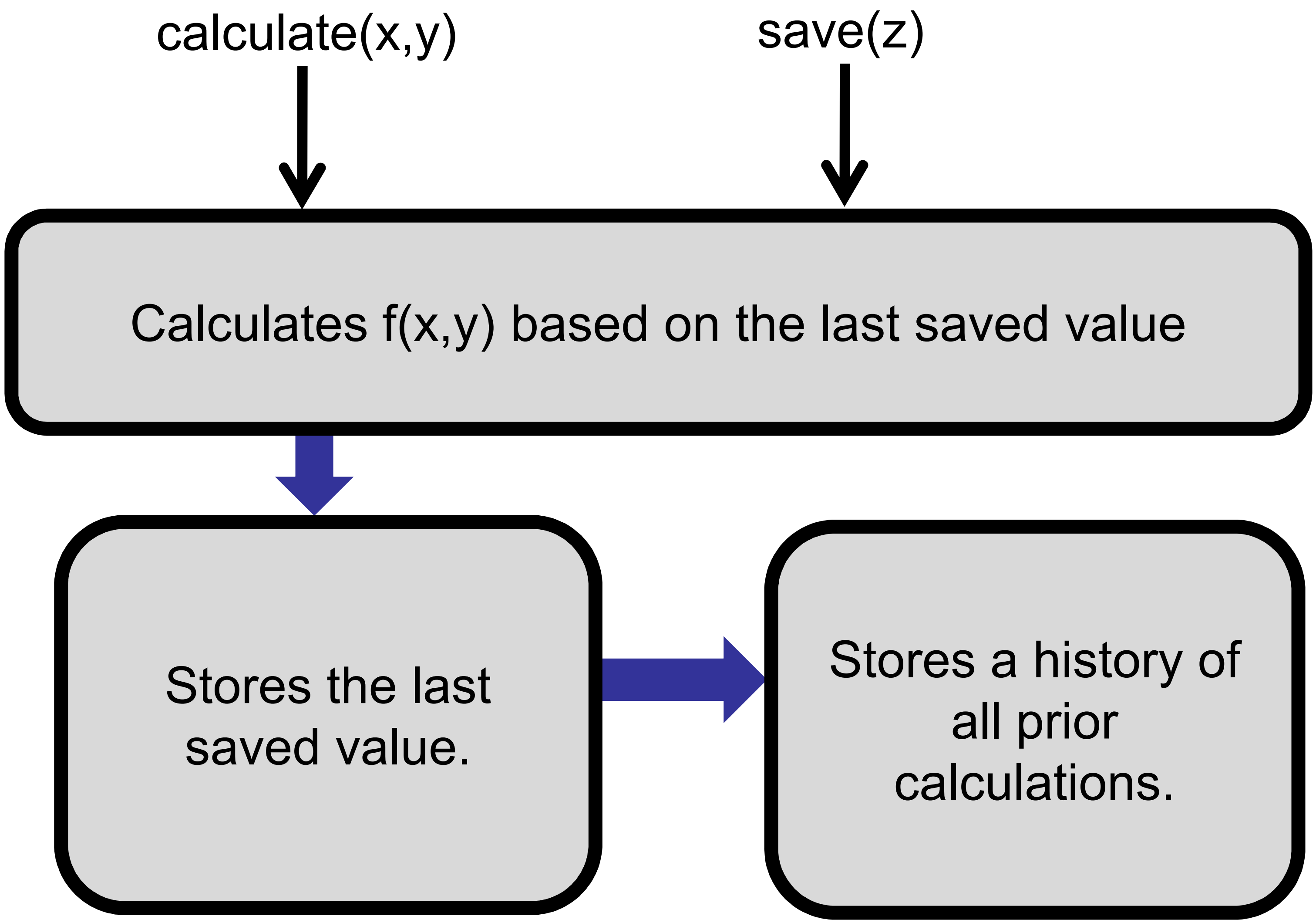
Abstraction



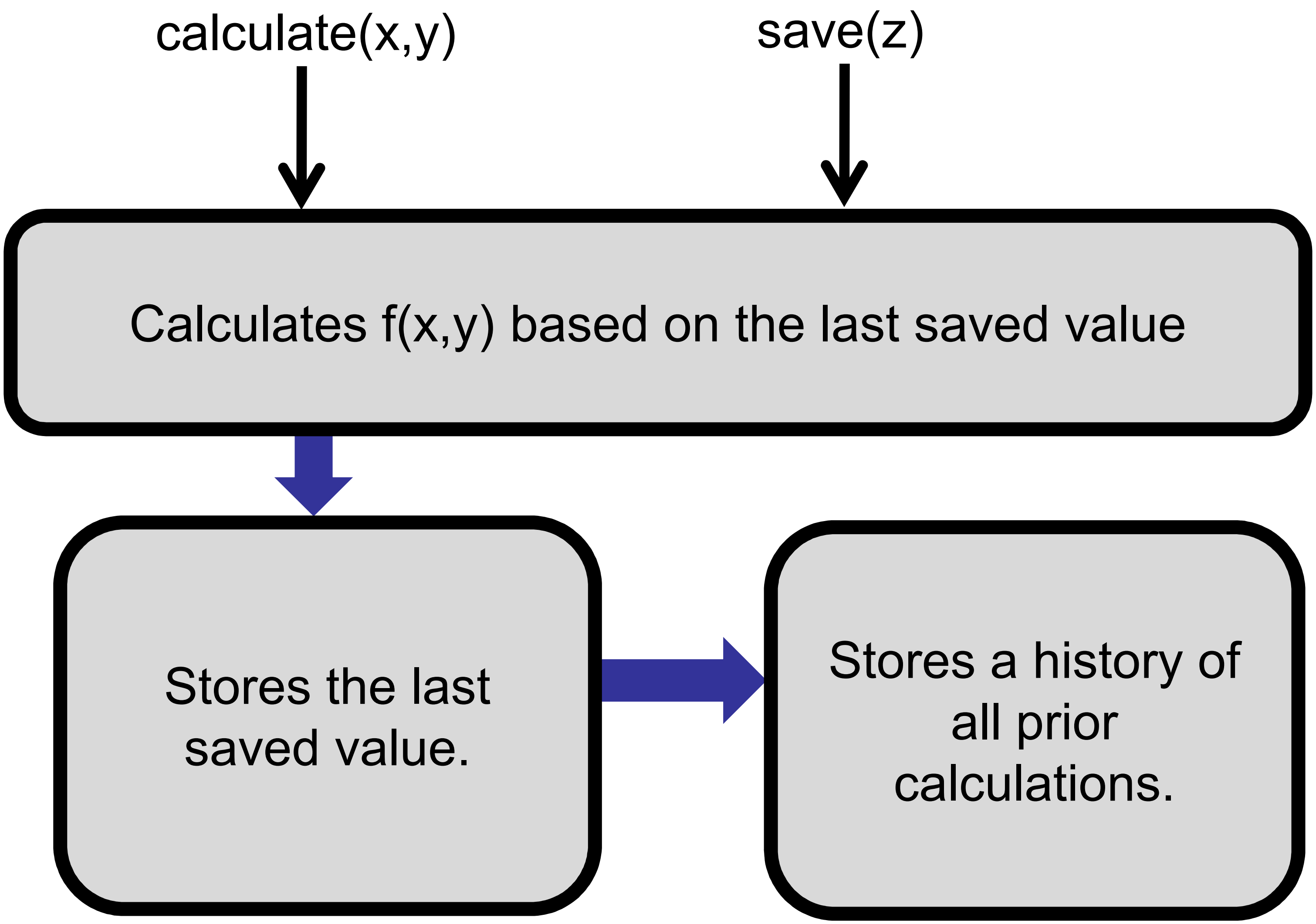
Abstraction



Abstraction



Top Down Design



Abstraction

Software engineering

- Divide problem into components.
- Define *interface* between components.
- Assign one team to build each component.
- (Recurse.)

- Top down design: get the big idea first, then figure out how to implement it.

Abstraction

Algorithm design

- Divide problem into components.
- Define *interface* between components.
- Solve each problem separately.
- (Recurse.)
- Combine solutions.

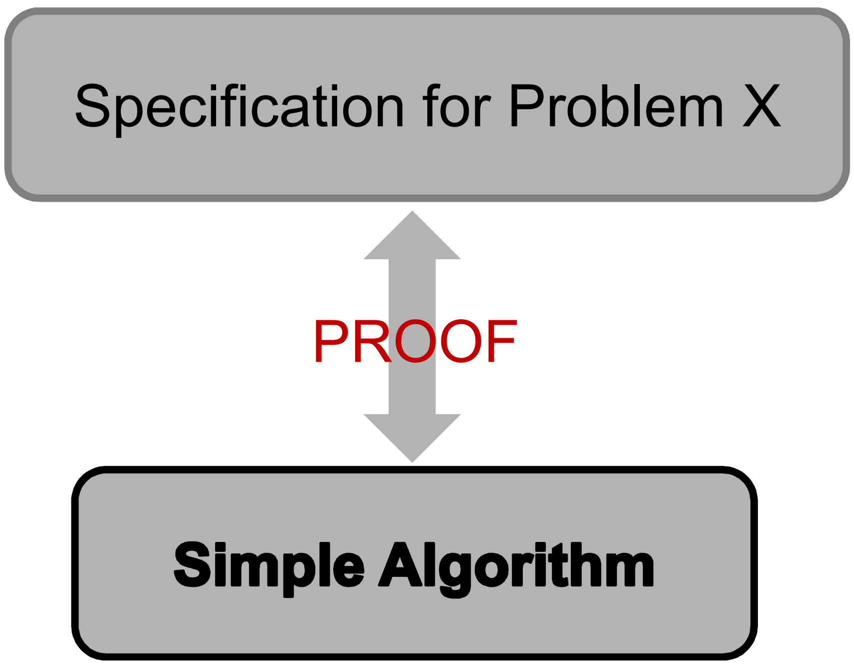
Abstraction

Algorithm design: divide-and-conquer

- Define sub-problems.
- State properties held by sub-problems.
- Solve sub-problems.
- (Recurse.)
- Combine solutions.

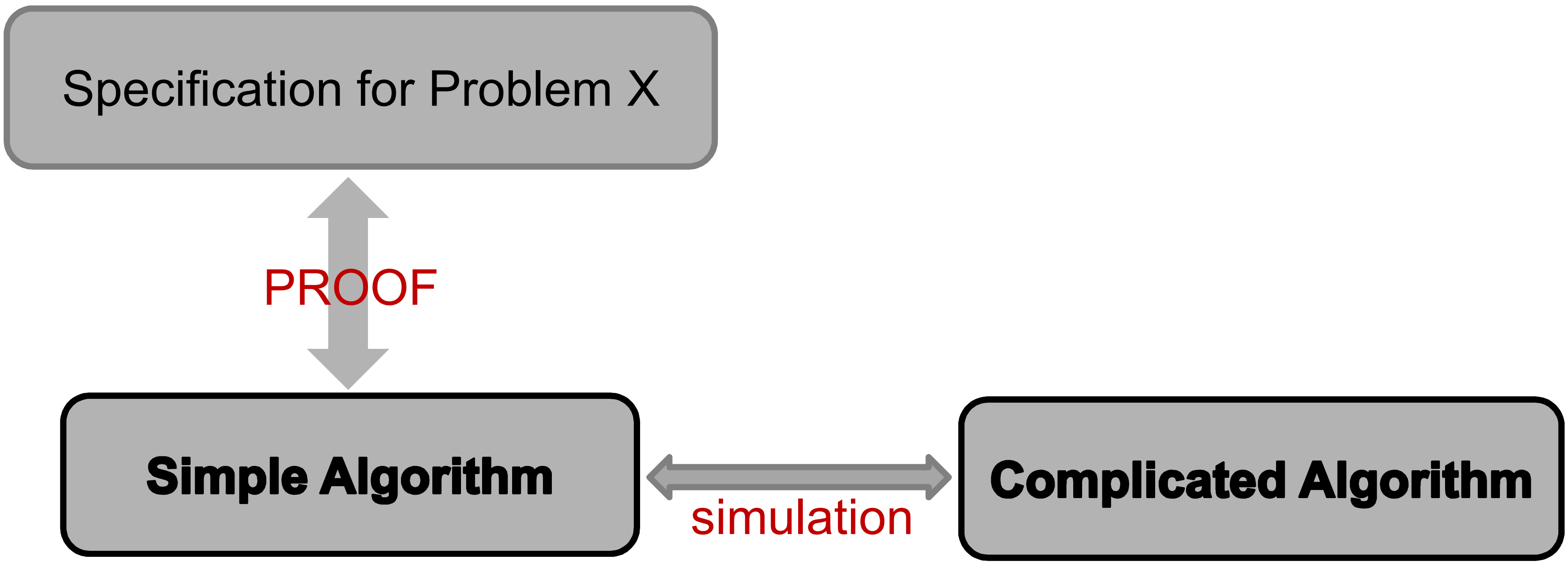
Abstraction

Algorithm design: iterated proofs



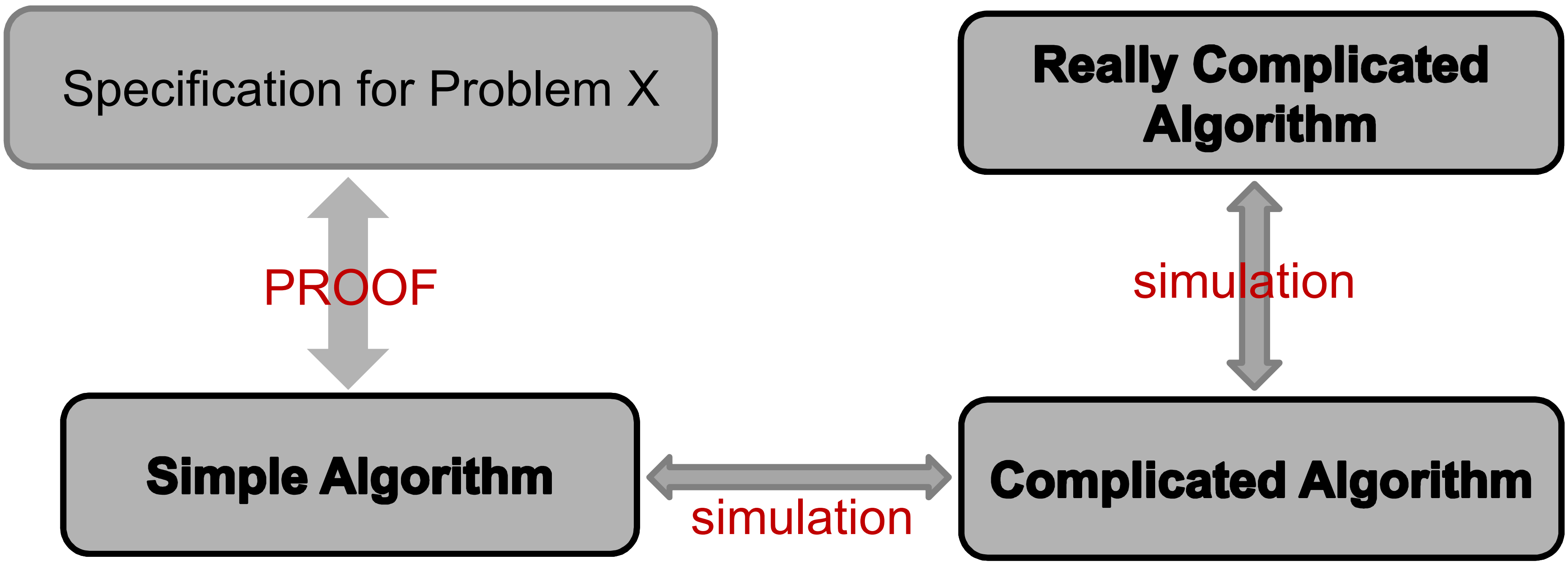
Abstraction

Algorithm design: iterated proofs



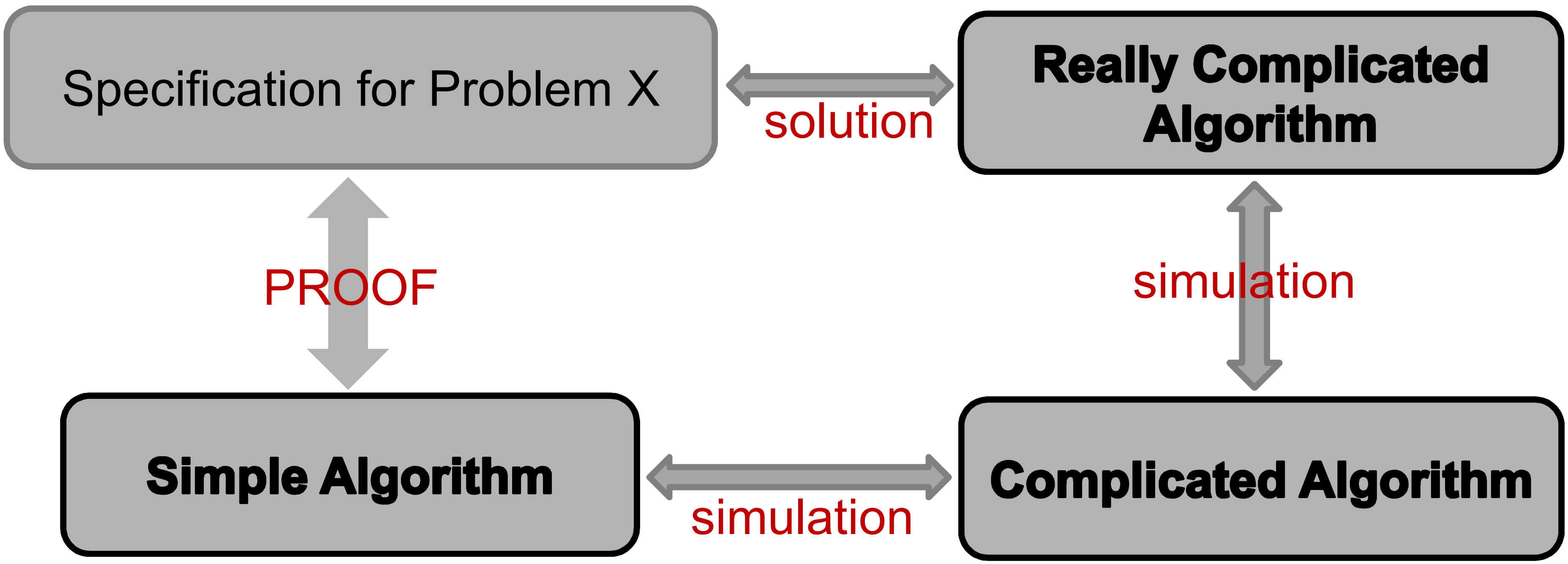
Abstraction

Algorithm design: iterated proofs



Abstraction

Algorithm design: iterated proofs



Abstraction

Key advantages

- Separate interface and implementation
- Hide implementation details
- Modularity: implement/analyze components separately

Abstract Data Types

Specification:

- Interface
- Behavior

Abstract Data Types

Specification:

- Interface
- Behavior

function: call(name)
returns: connection

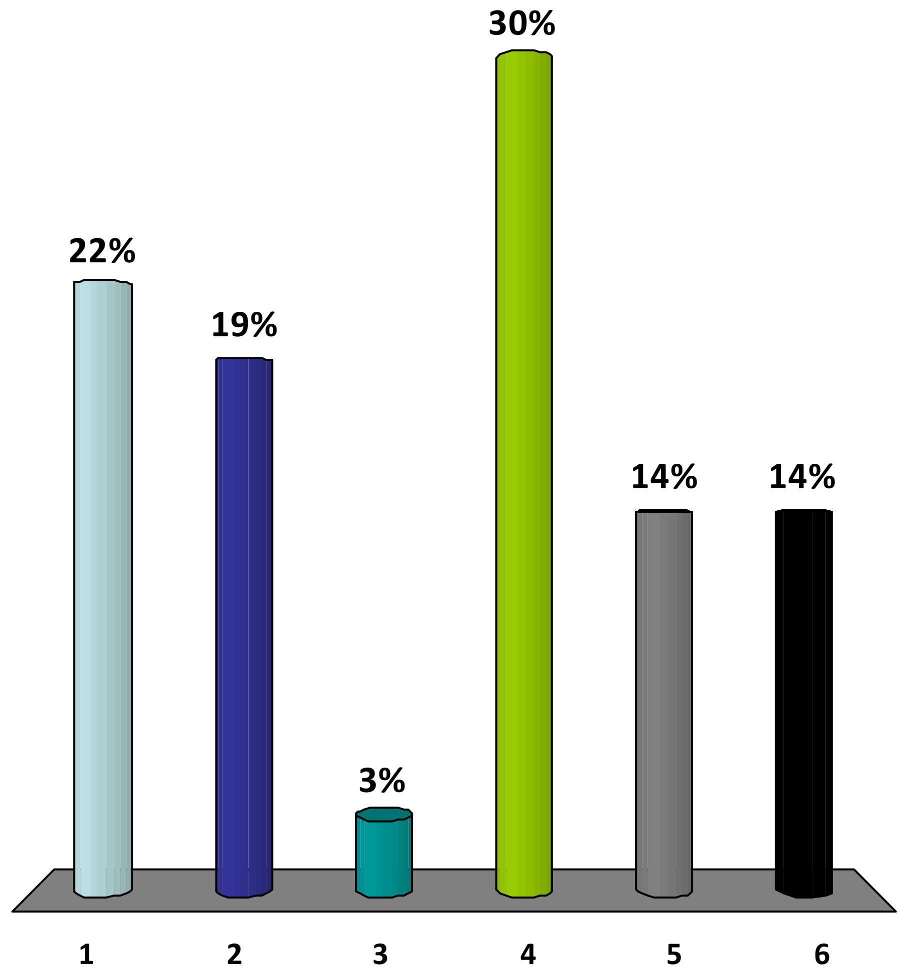
function: getPie(type)
returns: slice_of_pie

call(name) : Accesses 3G network and initiates a telephone call to name. Returns a connection object.

getPie(type) : Accesses bakery network and orders a pie of the specified type. Returns a slice_of_pie accessor.

My favorite type of phone is:

1. iPhone
2. Android
3. Windows Phone 7
4. Basic, simple functional phone.
5. Wired landline.
6. I don't use phones.



Abstract Data Types

Stack

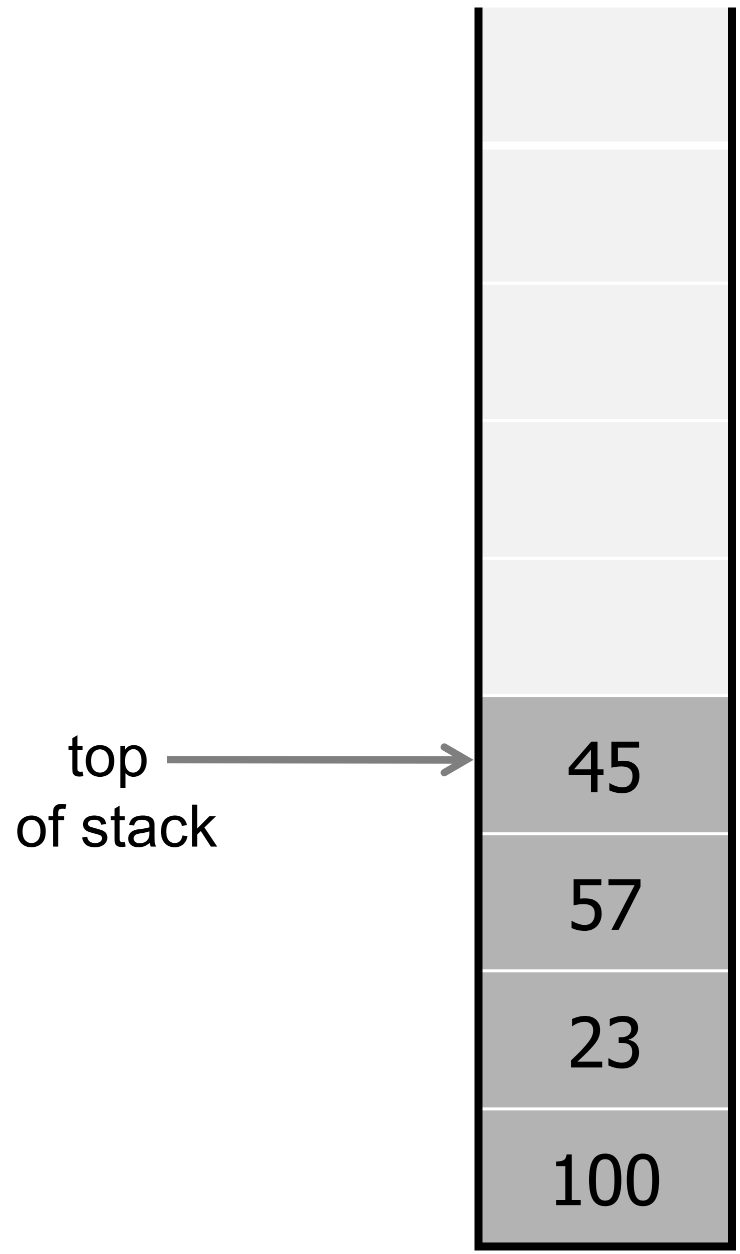
- Interface:
  - void push(element x)
  - element pop()

- Behavior: (LIFO: last-in, first-out)
  - push(x) : adds element x to the stack
  - pop() : removes the mostly recently added element and returns it

Abstract Data Types

Stack

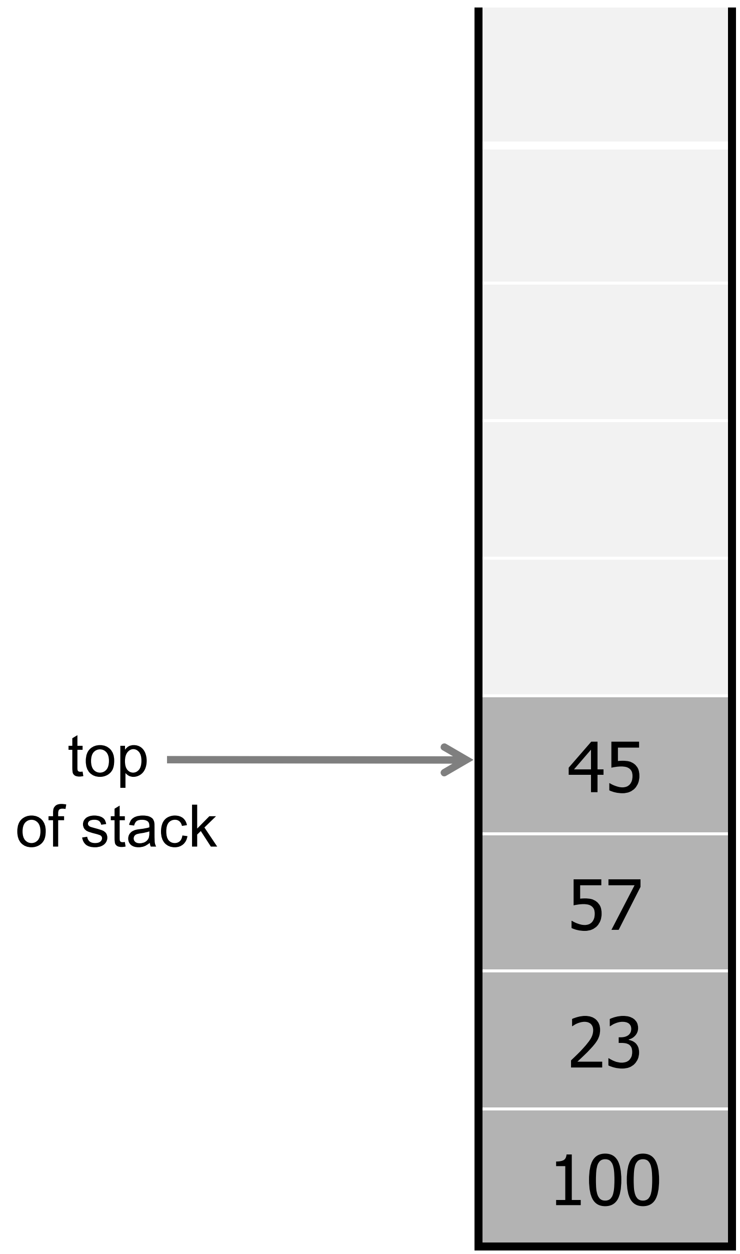
- Interface:
  - void push(element x)
  - element pop()
  - empty()



Abstract Data Types

Stack

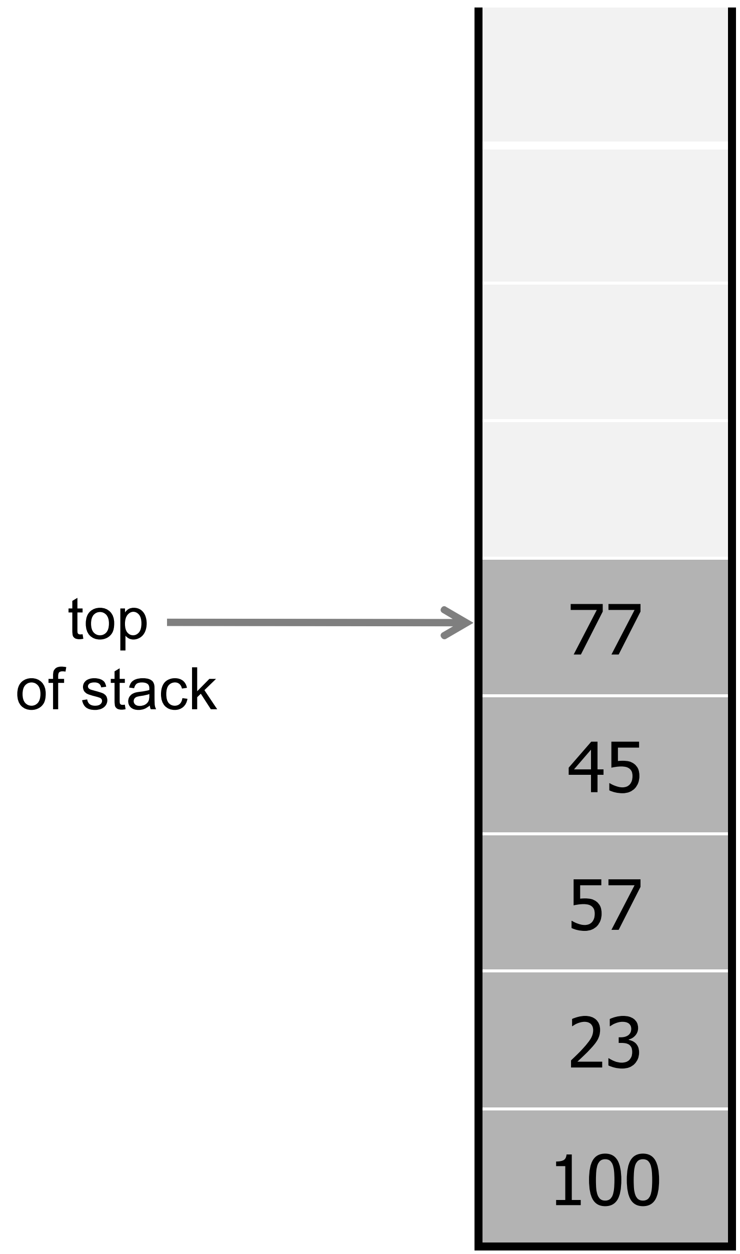
- Execution:
  - push(77)



Abstract Data Types

Stack

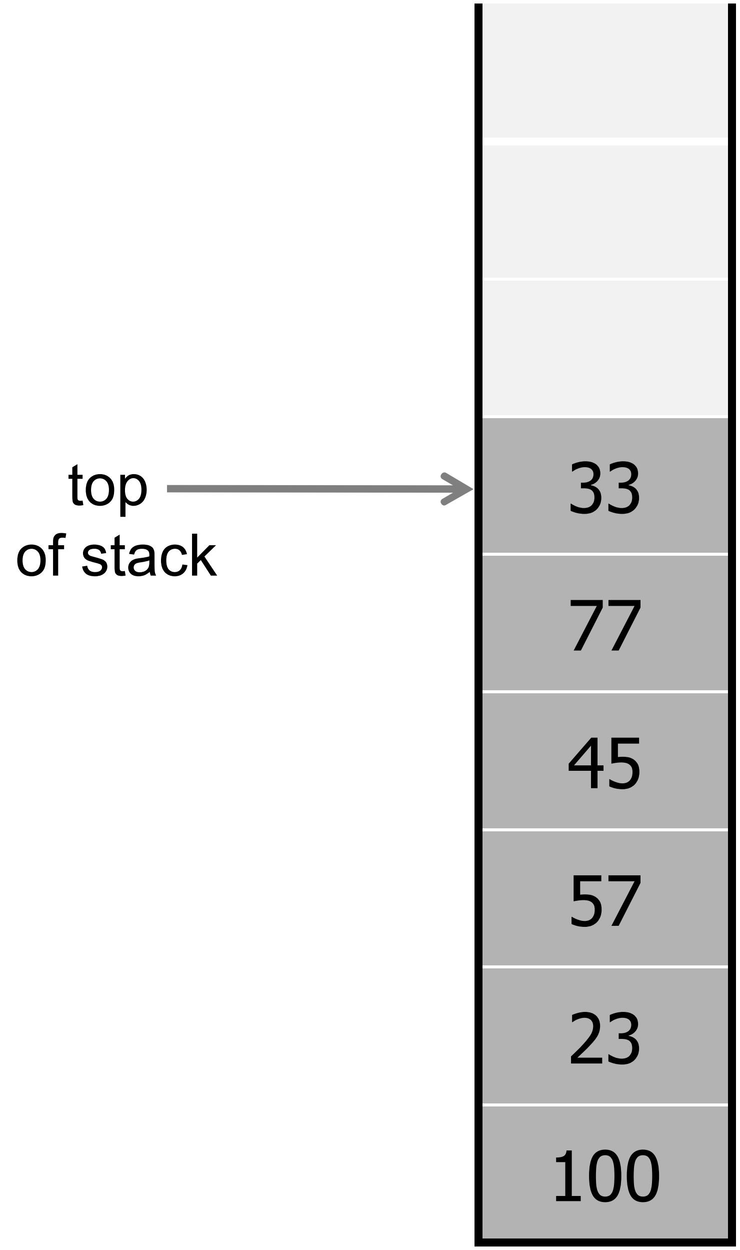
- Execution:
  - push(77)



Abstract Data Types

Stack

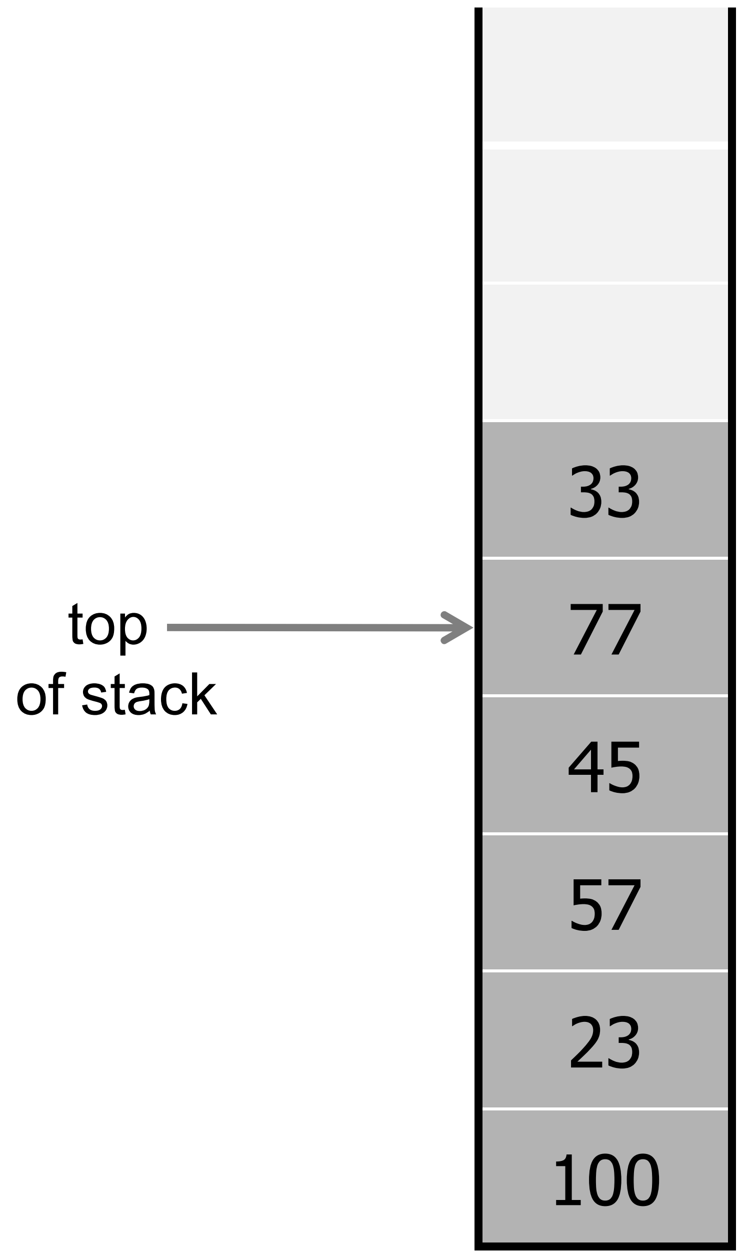
- Execution:
  - push(77)
  - push(33)



Abstract Data Types

Stack

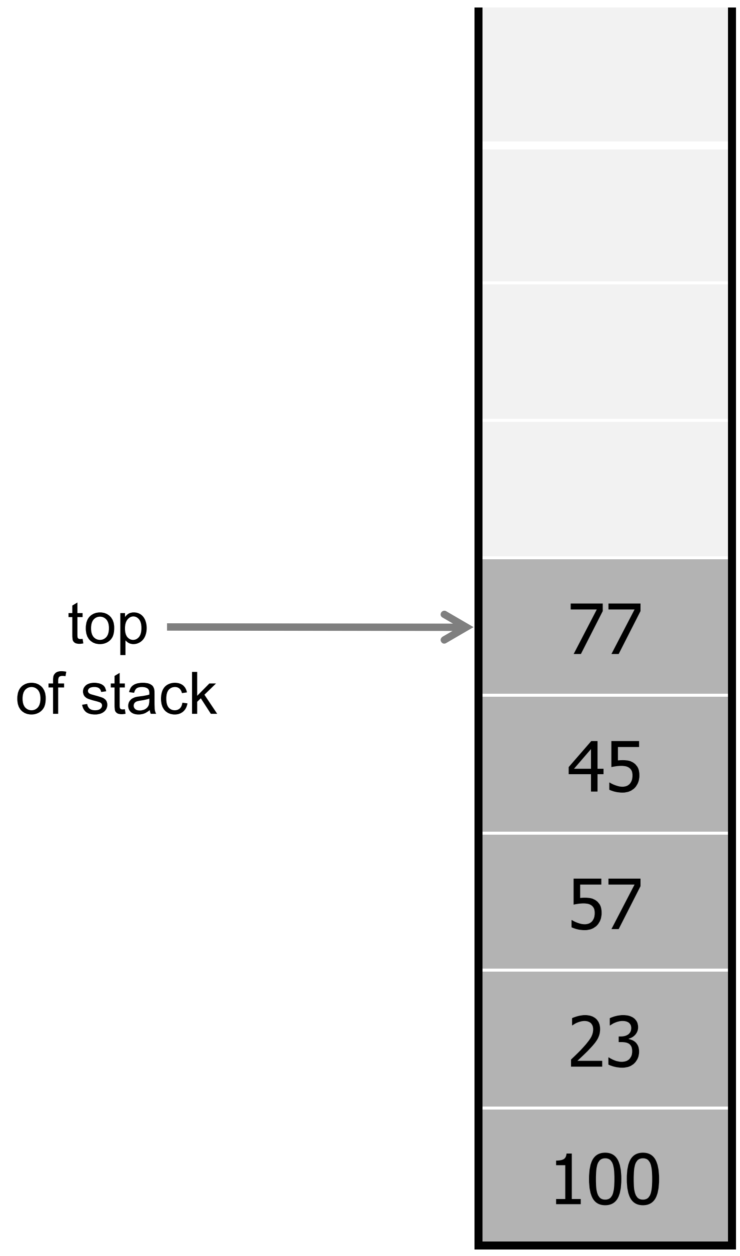
- Execution:
  - push(77)
  - push(33)
  - pop() → ??



Abstract Data Types

Stack

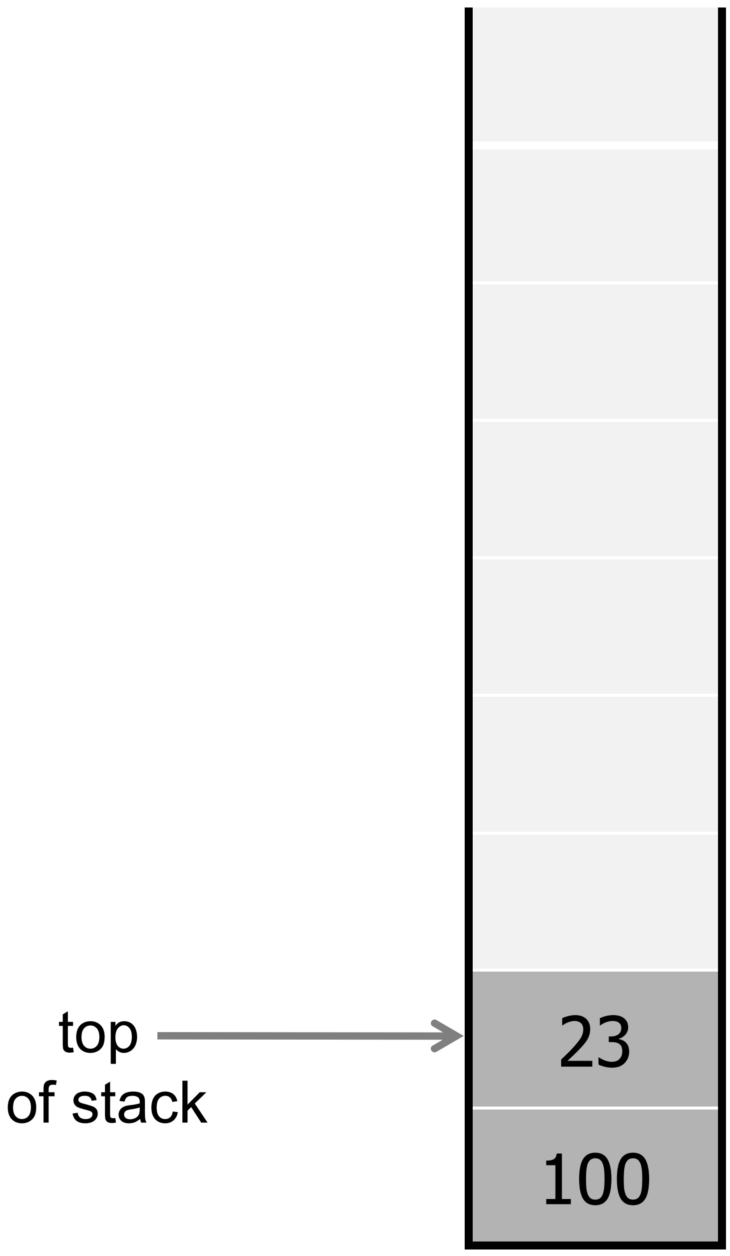
- Execution:
  - push(77)
  - push(33)
  - pop() → 33



Abstract Data Types

Stack

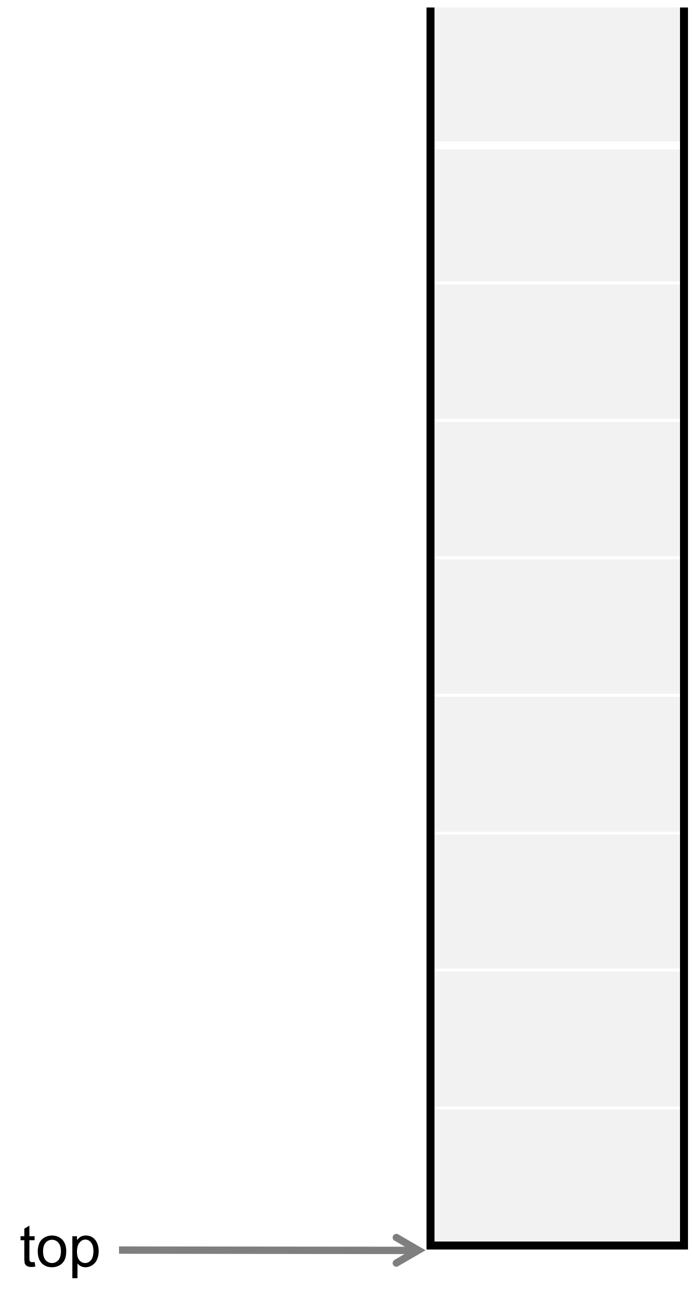
- Execution:
  - push(77)
  - push(33)
  - pop() → 33
  - pop() → 77
  - pop() → 45
  - pop() → 57



Abstract Data Types

Stack

- Execution:
  - pop() → 23
  - pop() → 100

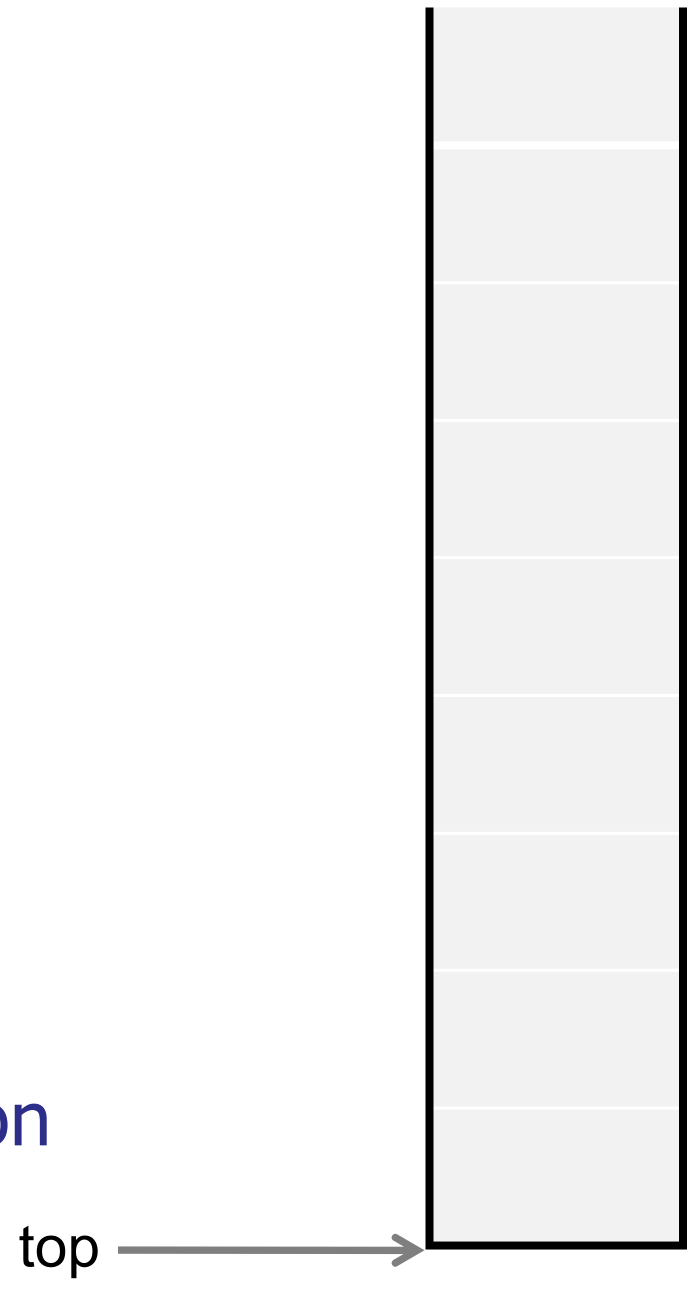


Abstract Data Types

Stack

- Execution:
  - pop() $\rightarrow$ 23
  - pop() $\rightarrow$ 100
  - pop() $\rightarrow$??

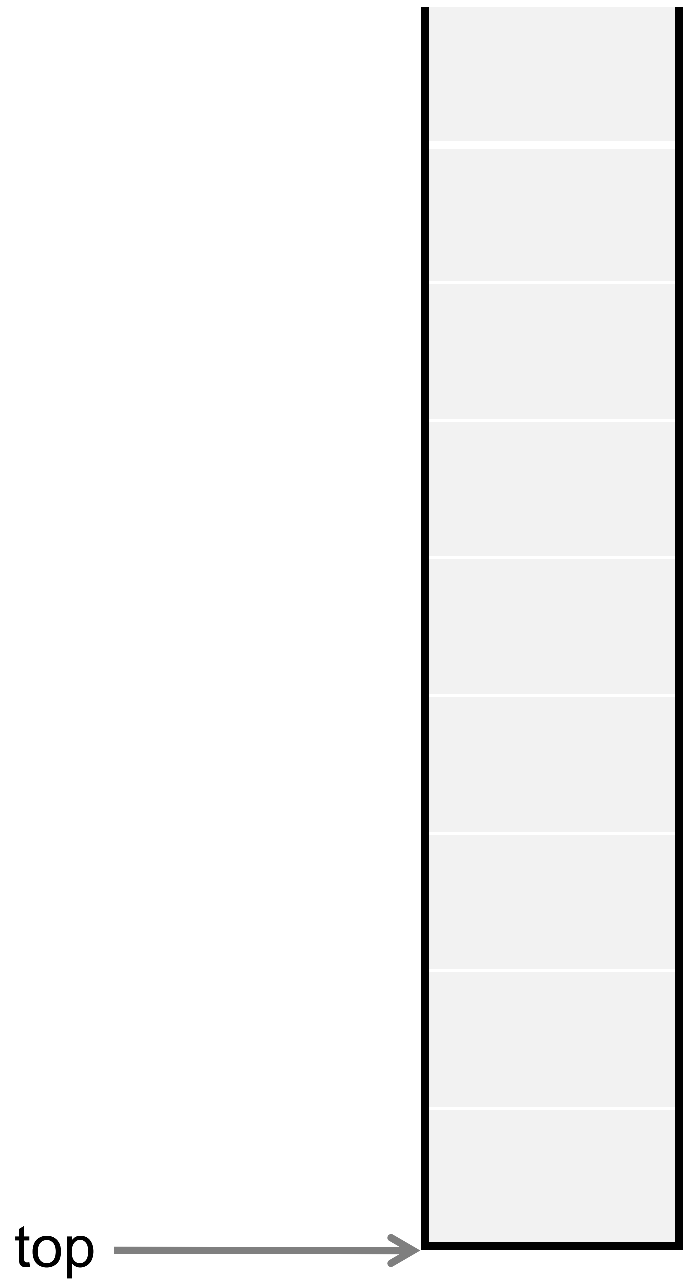
- Error!
  - Option 1: throw exception
  - Option 2: modify specification



Abstract Data Types

Stack

- Execution:
  - pop() → 23
  - pop() → 100
  - empty() → true



Abstract Data Types

Stack (of integers) Implementation:

```
class Stack{
    int[1000] stackArray;
    int top = 0;
```

```
boolean empty()
    return (top==0);
```

```
void push(int x)
    top++;
    stackArray[top] = x;
```

```
int pop()
    int i = stackArray[top];
    top--;
    return i;
```

Abstract Data Types

Stack (of integers) Implementation:

```
class Stack{
    int[1000] stackArray;
    int top = 0;
```

```
boolean empty()
    return (top==0);
```

What if stack is empty?

```
int pop()
    int i = stackArray[top];
    top--;
    return i;
```

```
void push(int x)
    top++;
    stackArray[top] = x;
```

Abstract Data Types

Stack (of integers) Implementation:

What if stack has 1001 elements?

```
class Stack{
    int[1000] stackArray;
    int top = 0;
```

```
boolean empty()
    return (top==0);
```

```
void push(int x)
    top++;
    stackArray[top] = x;
```

```
int pop()
    int i = stackArray[top];
    top--;
    return i;
```

Abstract Data Types

Stack (of integers) Implementation:

- Three solutions:
  - Return error on overflow.
  - Resize array on overflow.
  - Use auto-resizing array: java.util.ArrayList

Abstract Data Types in Java

Goal: hide the implementation

```
interface Stack {
    void push(int x);

    int pop() throws StackEmptyException;

    boolean empty();
}
```

Abstract Data Types in Java

```
class mySpecialStack  implements Stack{
    int[] stackArray;

    void push(int x){
          …
    }

    int pop() throws StackEmptyException{
          …
    }

    boolean empty(){
          …
    }
}
```

Abstract Data Types in Java

Using a stack:

```
void fillStack()
{
   mySpecialStack storeStack = new mySpecialStack;

   for (int i=0; i<1000; i++)
   {
        storeStack.push(i);
   }
}
```

Abstract Data Types in Java

Using a stack:

A mySpecialStack is a Stack.

```
void fillStack()
{
    Stack storeStack = new mySpecialStack;

    for (int i=0; i<1000; i++)
    {
        storeStack.push(i);
    }
}
```

Abstract Data Types in Java

Using a stack:

```
void fillStack(Stack storeStack)
{
   for (int i=0; i<1000; i++)
   {
        storeStack.push(i);
   }
}
```

```
{
   Stack A = new SlowStack
   fillStack(A);
}
```

```
{
   Stack B = new FastStack
   fillStack(B);
}
```

Abstract Data Types in Java

Generics:

```
interface Stack<TYPE> {
    void push(TYPE x);

    TYPE pop() throws StackEmptyException;

    boolean empty();
}
```

Abstract Data Types in Java

```
class mySpecialStack<T>  implements Stack<T>{
    T[] stackArray;

    void push(T x){
        …
    }

    T pop() throws StackEmptyException{
        …
    }

    boolean empty(){
        …
    }
}
```

Inheritance

What if I want to build a better Stack?

- Option 1: implement stack

```
class myBetterStack  implements Stack{
   // implement push, pop, and empty
   …
}
```

- Useful when:

  Entirely new implementation (e.g., don't use an array, use fractional cascading on a buffered tree).

Inheritance

What if I want to build a better Stack?

- Option 2: extend old implementation

```
class myBetterStack  extends SlowStack{
   // Only implement new version of empty()
   boolean empty(){
          …
   }
}
```

- Useful when:

  Building a new version of an existing object.

Inheritance

Extending an existing class

- Can re-implement (*override*) existing methods.
- Can add new methods.
- Can develop new functionality.

Abstract Data Types

Queue

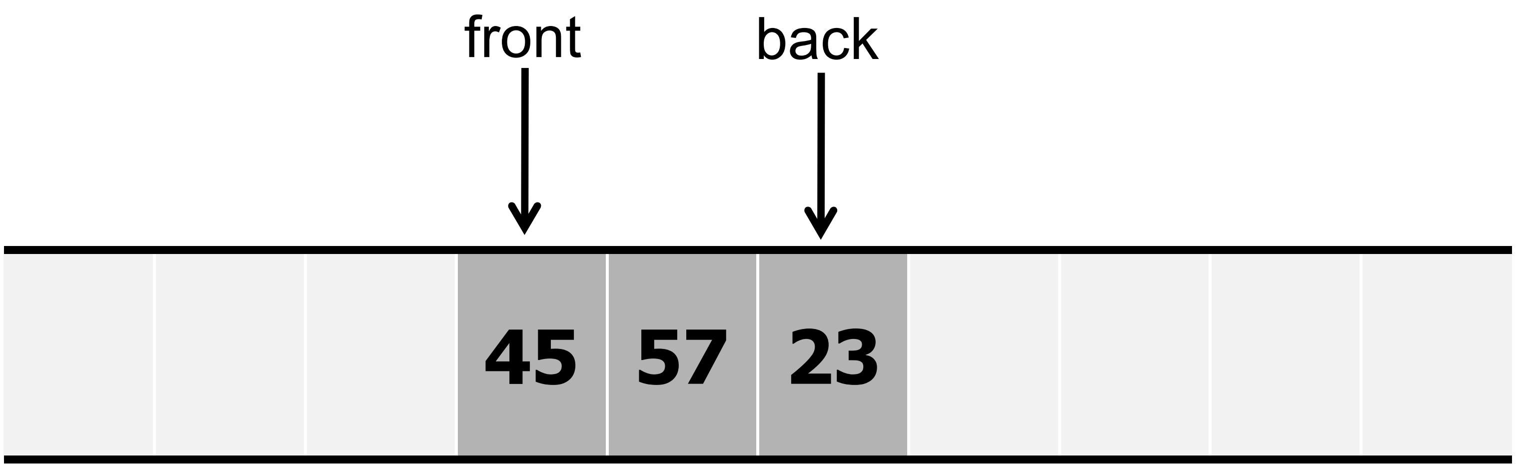
- Interface:
  - void enqueue(element x)
  - element dequeue()

- Behavior: (FIFO: last-in, first-out)
  - enqueue(x) : adds element x to the front of the queue
  - dequeue() : removes and returns element at the end of the queue

Abstract Data Types

Queue

Execution:

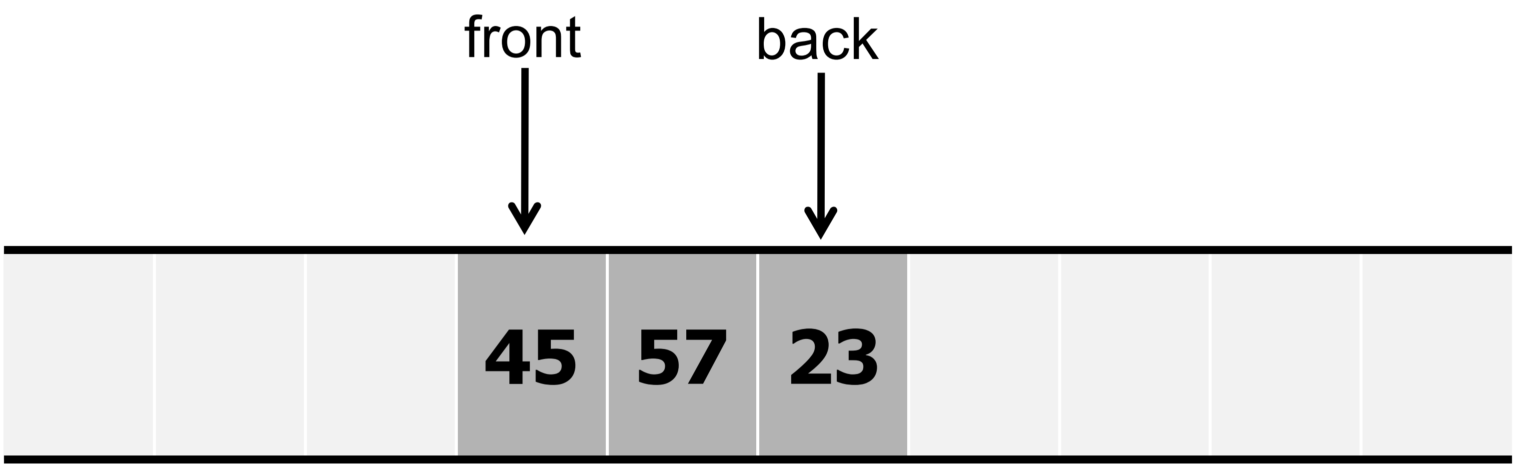


Abstract Data Types

Queue

Execution:

- enqueue(7)

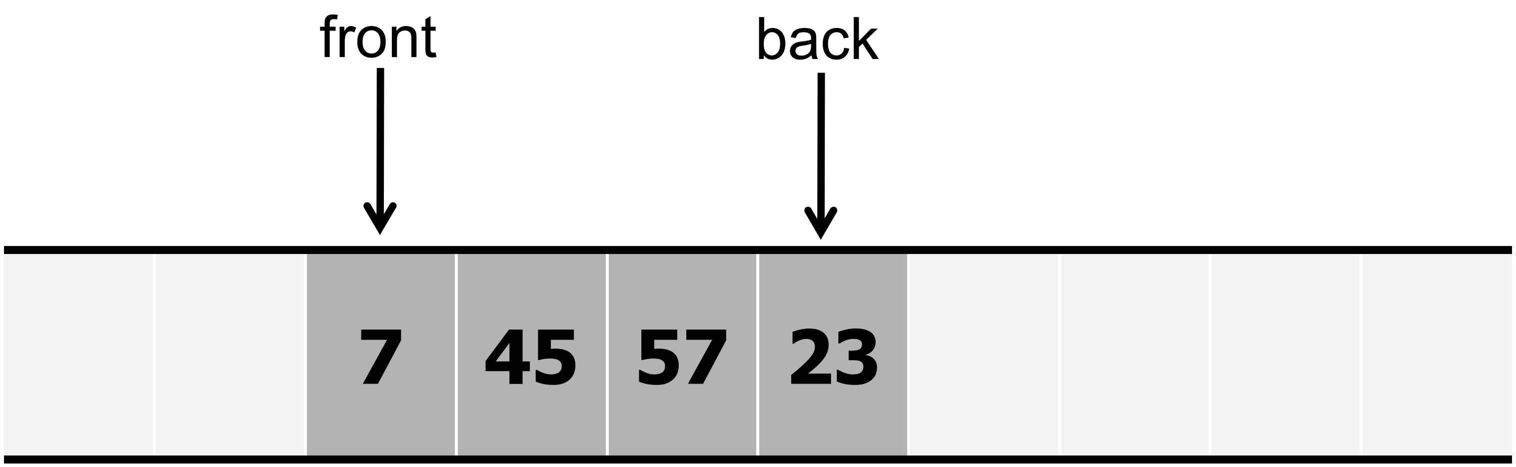


Abstract Data Types

Queue

Execution:

- enqueue(7)

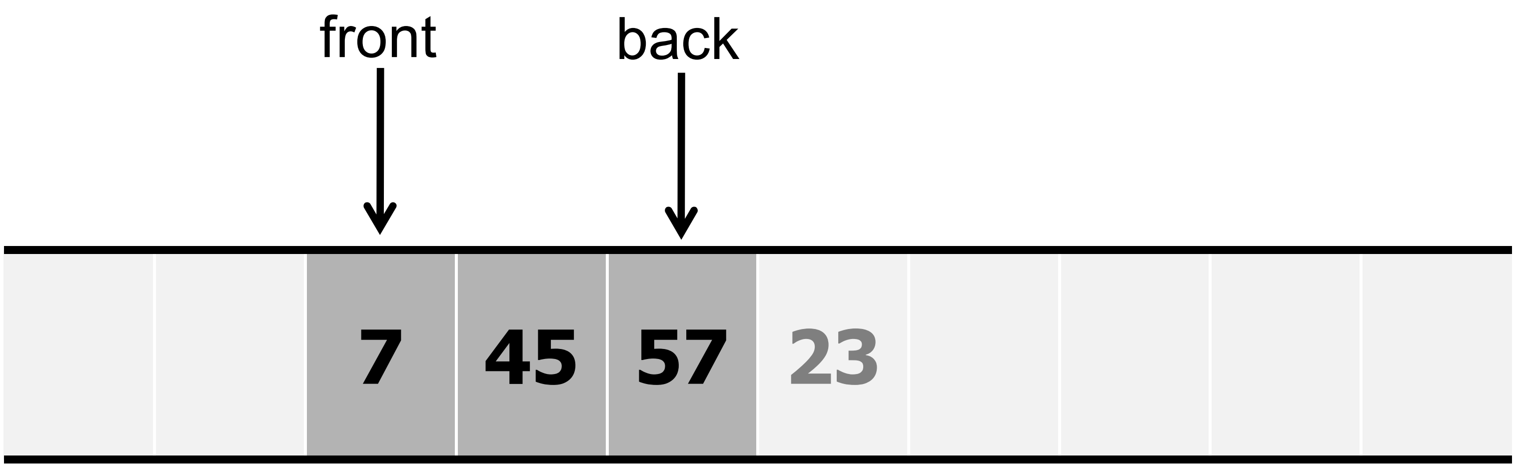


Abstract Data Types

Queue

Execution:

- enqueue(7)
- dequeue() → 23



Inheritance

Queue interface:

```
interface Queue{
   void enqueue(int x);

   int dequeue() throws QueueEmptyException;

   boolean empty();
}
```

Inheritance

Queue implementation:

```
class SimpleQueue implements Queue{
    void enqueue(int x){…}

    int dequeue() throws QueueEmptyException{ … }

    boolean empty(){…}
}
```

Inheritance

Better Queue implementation:

```
class BetterQueue extends SimpleQueue{
    int size(){…}
}
```

Inheritance

Better Queue implementation:

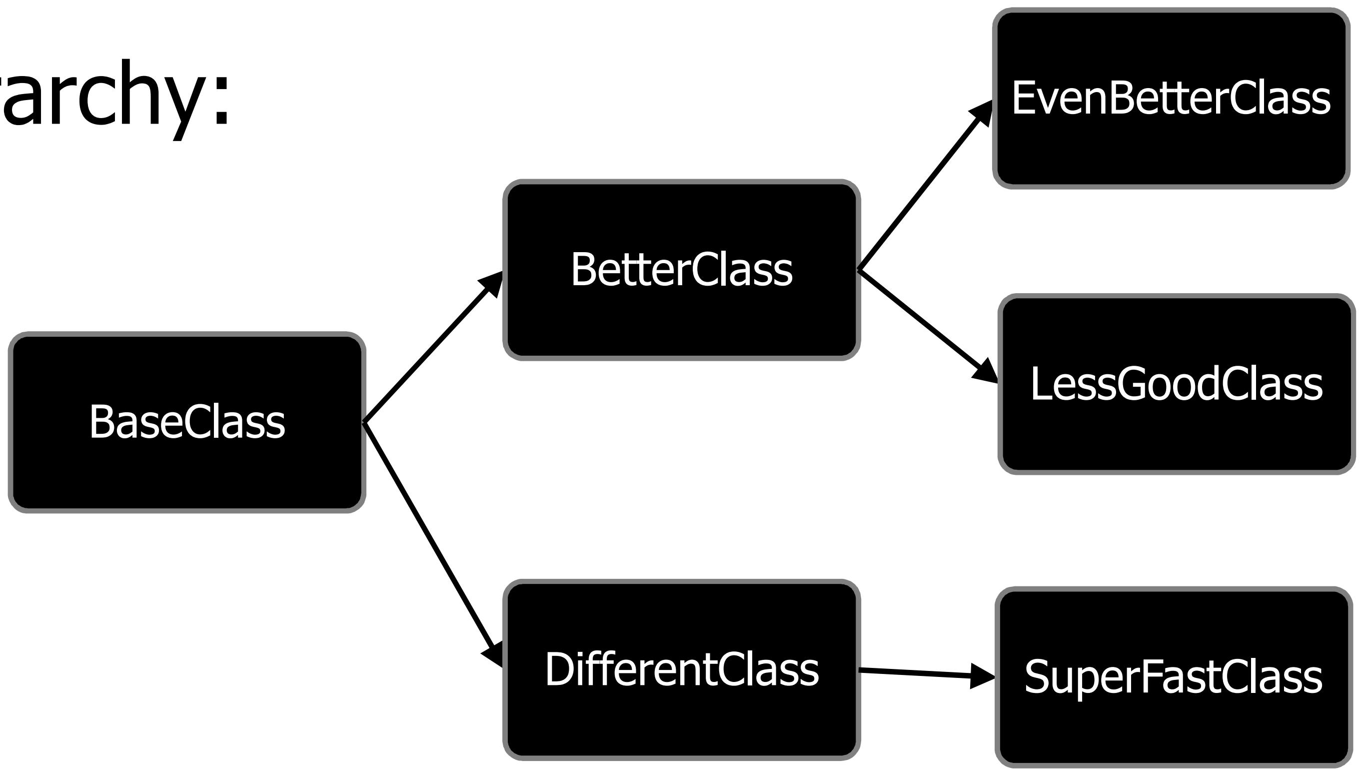
```
class StackQueue extends SimpleQueue implements Stack{
   int push(int x){…}

   int pop() throws EmptyStackException {…}

   // no need to implement the queue, since inherited
}
```

Inheritance

Rules of inheritance:

- You can implement many interfaces.
- You can only extend one class.

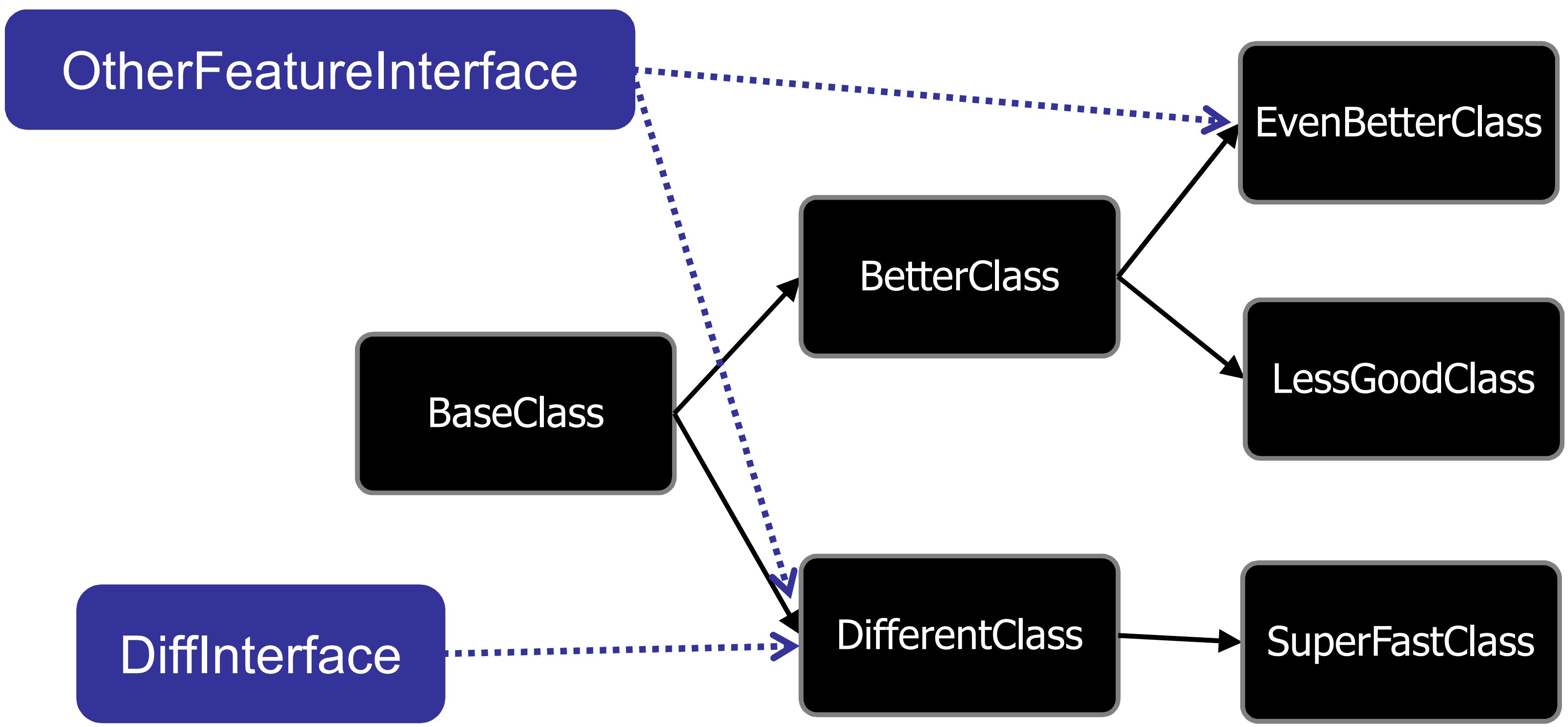
Class hierarchy:



Inheritance

Rules of inheritance:

- You can implement many interfaces.
- You can only extend one class.



Inheritance

VectorTextFile class:

- v1: slow
- v2: improved string management
- v3: improved sorting
- v3.5: Problem Set 2
- v4: no sorting

Problem:

- How to figure out what changed from v2 to v3?

Inheritance

VectorTextFile class:

- v1: slow
- v2: improved string management
- v3: improved sorting
- v3.5: Problem Set 2
- v4: no sorting

Good practice:

- Use inheritance!
- Each version contains only what is new.

Break time

Questions?

Come talk to me about recitation scheduling.

Recitations

Three time slots:

- 1pm
- 2pm
- 4pm

Today:

- If you are scheduled for one of these slots already, come to that one.
- If you are scheduled for 3pm, choose one of the others (preferably 1pm or 2pm).

Airport Scheduling

Simple Runway Problem:

- Small airport (not Changi) has 1 runway.
- Airplanes want to land:
  - Given: requested landing time
  - Requirement: 3 minutes between planes
  - Output: yes/no

Harder Airport Scheduling

Multiple Runway Problem:

- Changi airport has k runways.
- Airplanes want to land:
  - Given: requested landing time
  - Requirement: 3 minutes between planes
  - Output: yes/no

Not today…

- Think of scheduling computing jobs on a network.

Airport Scheduling

Simple Runway Problem:

- Small airport (not Changi) has 1 runway.
- Airplanes want to land:
  - Given: requested landing time
  - Requirement: 3 minutes between planes
  - Output: yes/no

```
interface Runway{
    // return true if scheduled for time t
    // return false if scheduling fails
    boolean requestLanding(time t);
}
```

Airport Scheduling

Simple Runway Problem:

- Small airport (not Changi) has 1 runway.
- Airplanes want to land:
  - Given: requested landing time
  - Requirement: 3 minutes between planes
  - Output: yes/no
- Additional requirements:
  - How many planes scheduled between: 9:00-11:00am?
  - Cancel landing reservation.

Airport Scheduling

Implementing ideas?

Airport Scheduling

Algorithm 1:

- Maintain a list of landing times.

7:00	**6:35**	**14:23**	**12:21**	**7:19**	**8:21**	**14:42**			

- On a request for time t, scan the list.
- If time t is safe, then add t to the end of the list.

7:00	**6:35**	**14:23**	**12:21**	**7:19**	**8:21**	**14:42**	t		

Airport Scheduling

Implementing ideas?

Airport Scheduling

Algorithm 2:

- Maintain a **sorted** list of landing times.

6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

- On a request for time t, binary search the list.
- If time t is safe, then add t to the end of the list.

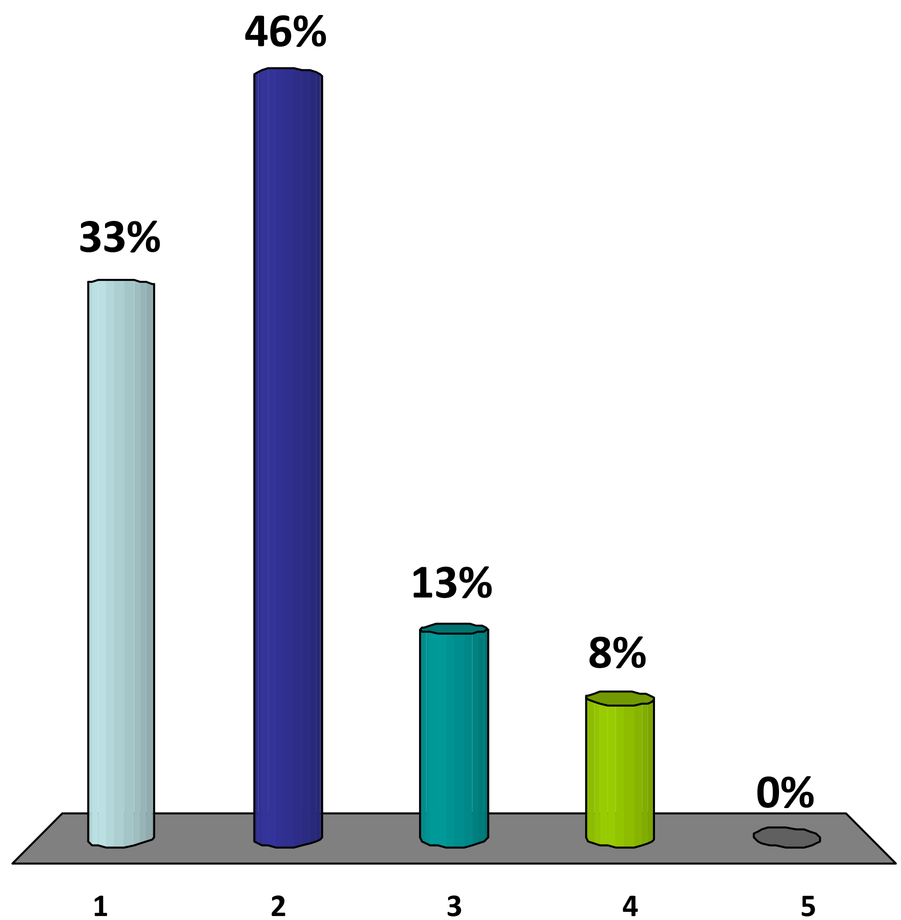
6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**	t		

- Re-sort:

6:35	**7:00**	**7:19**	t	**8:21**	**12:21**	**14:23**	**14:42**		

Running time for Algorithm 2:

1. O(log n)
2. O(n)
3. O(n log n)
4. O(n^2)
5. O(2^n)



Airport Scheduling

Algorithm 2b:

- Maintain a **sorted** list of landing times.

6:35	7:00	7:19	8:21	12:21	14:23	14:42			

- On a request for time t, binary search the list.
- If time t is safe, then make space for t by moving other times over.

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		

Running time: O(n)

Airport Scheduling

Algorithm 3:

- Maintain a list of all times.

...	7:00	7:01	7:02	7:03	7:04	7:05	7:06	7:07	...
		X			?			X	

- If times: [t-2, t-1, t, t+1, t+2] are free, schedule plane.

Running time: O(1)

Space: (24*60)

What if arrival times are not on-the-minute?

Airport Scheduling

Algorithm 3:

- Maintain a list of all times.

...	7:00	7:01	7:02	7:03	7:04	7:05	7:06	7:07	...
		X			?			X	

- If times: [t-2, t-1, t, t+1, t+2] are free, schedule plane.

Problems:

What if arrival times are not on-the-minute?

Expensive to calculate number of scheduled planes.

Airport Scheduling

Dynamic Dictionary

- Interface:
  - void insert(time t)
  - boolean search(time t)
  - time successor(time t)
  - time predecessor(time t)

Airport Scheduling

Dynamic Dictionary

- Interface:
  - void insert(time t)
  - boolean search(time t)
  - time successor(time t)
  - time predecessor(time t)

Adds new item to dictionary.

Airport Scheduling

Dynamic Dictionary

- Interface:
  - void insert(time t)
  - boolean search(time t)
  - time successor(time t)
  - time predecessor(time t)

Searches for item in dictionary.

Airport Scheduling

Dynamic Dictionary

- Interface:
  - void insert(time t)
  - boolean search(time t)
  - time successor(time t)
  - time predecessor(time t)

Find first item in dictionary that is bigger than **t**.

Airport Scheduling

Dynamic Dictionary

- Interface:
  - void insert(time t)
  - boolean search(time t)
  - time successor(time t)
  - time predecessor(time t)

Find biggest item in dictionary that is smaller than **t**.

Airport Scheduling

Dynamic Dictionary

6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**	**12**		

- insert(t)

6:35	**7:00**	**7:19**	**t**	**8:21**	**12:21**	**14:23**	**14:42**		

Airport Scheduling

Dynamic Dictionary

6:35	7:00	7:19	8:21	12:21	14:23	14:42			

- insert(t)

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		

- search(8:24) $\rightarrow$ **false**
- search(8:21) $\rightarrow$ **true**

Airport Scheduling

Dynamic Dictionary

6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

- search-successor(8:24) = 12:21

Airport Scheduling

Dynamic Dictionary

6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

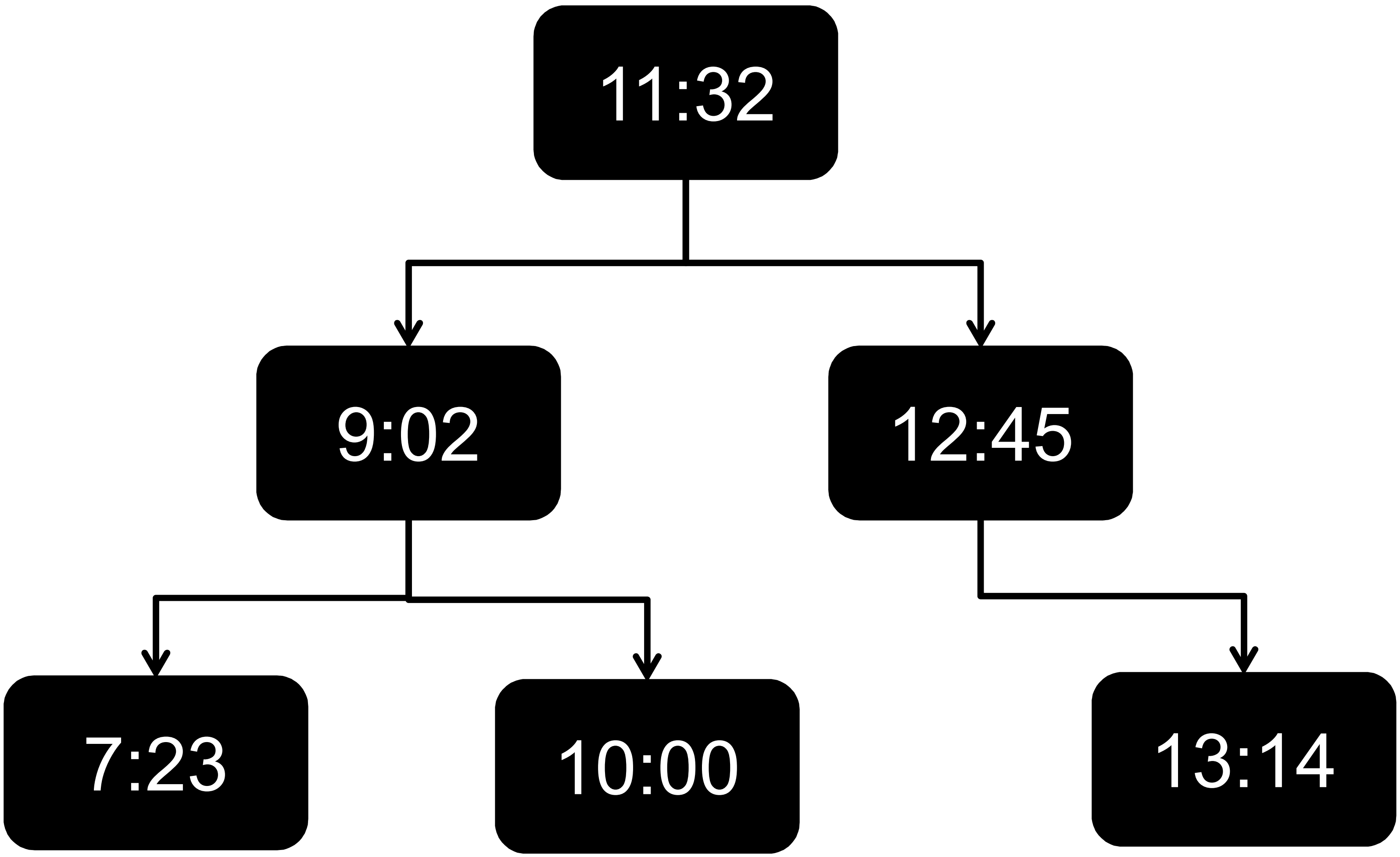
– search-predecessor(14:41) = 14:23

Airport Scheduling

```
class SimpleRunway implements Runway{
   DynamicDictionary dict;

   boolean requestLanding(time t){
        if ((!dict.search(t)) &&
            (t – dict.search-predecessor(t) > 3) &&
            (dict.search-sucessor(t) – t > 3))
        {
              dict.insert(t);
              return true;
        }
        return false;
   }
}
```

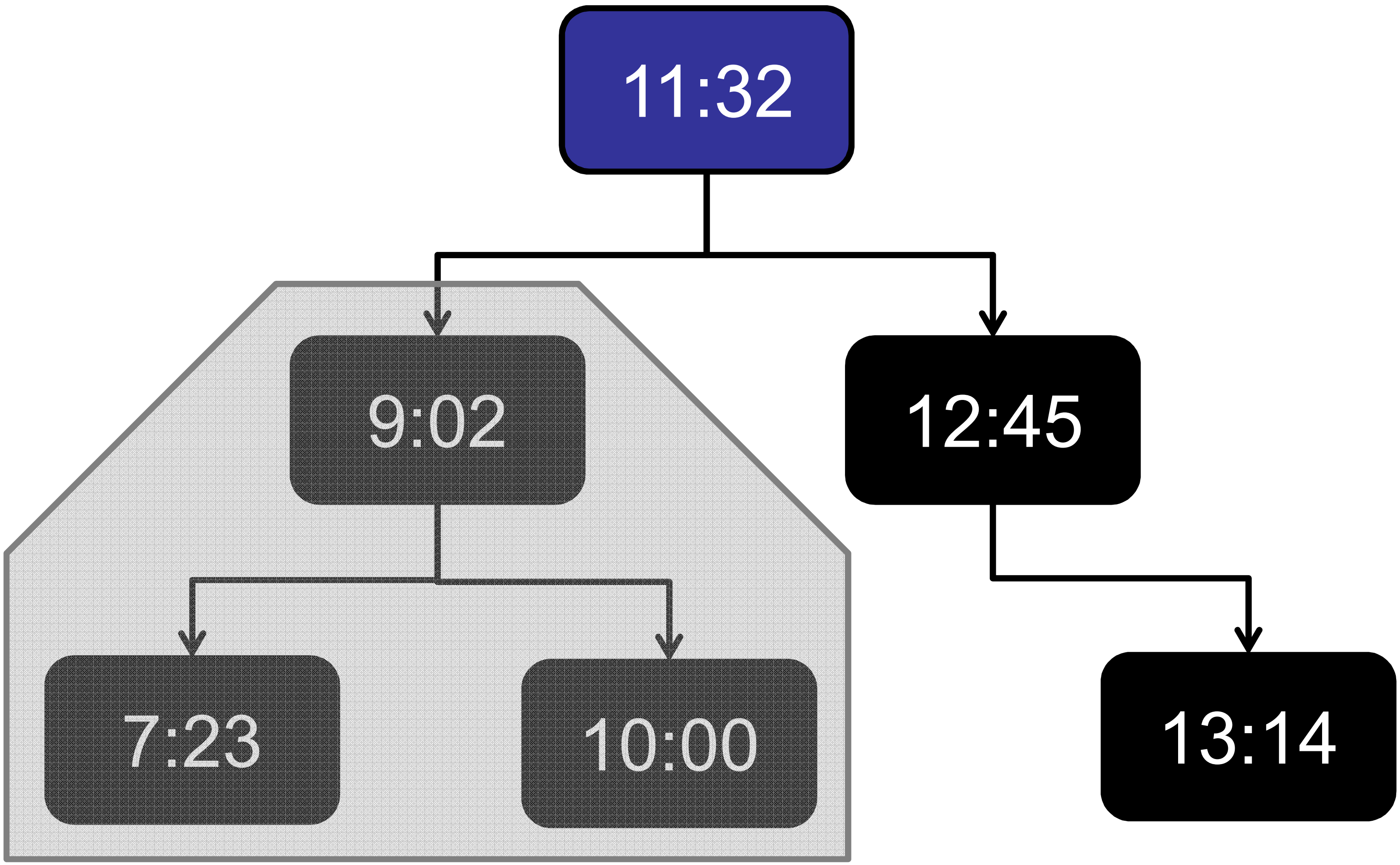
Dynamic Dictionary

Implementation idea: Tree



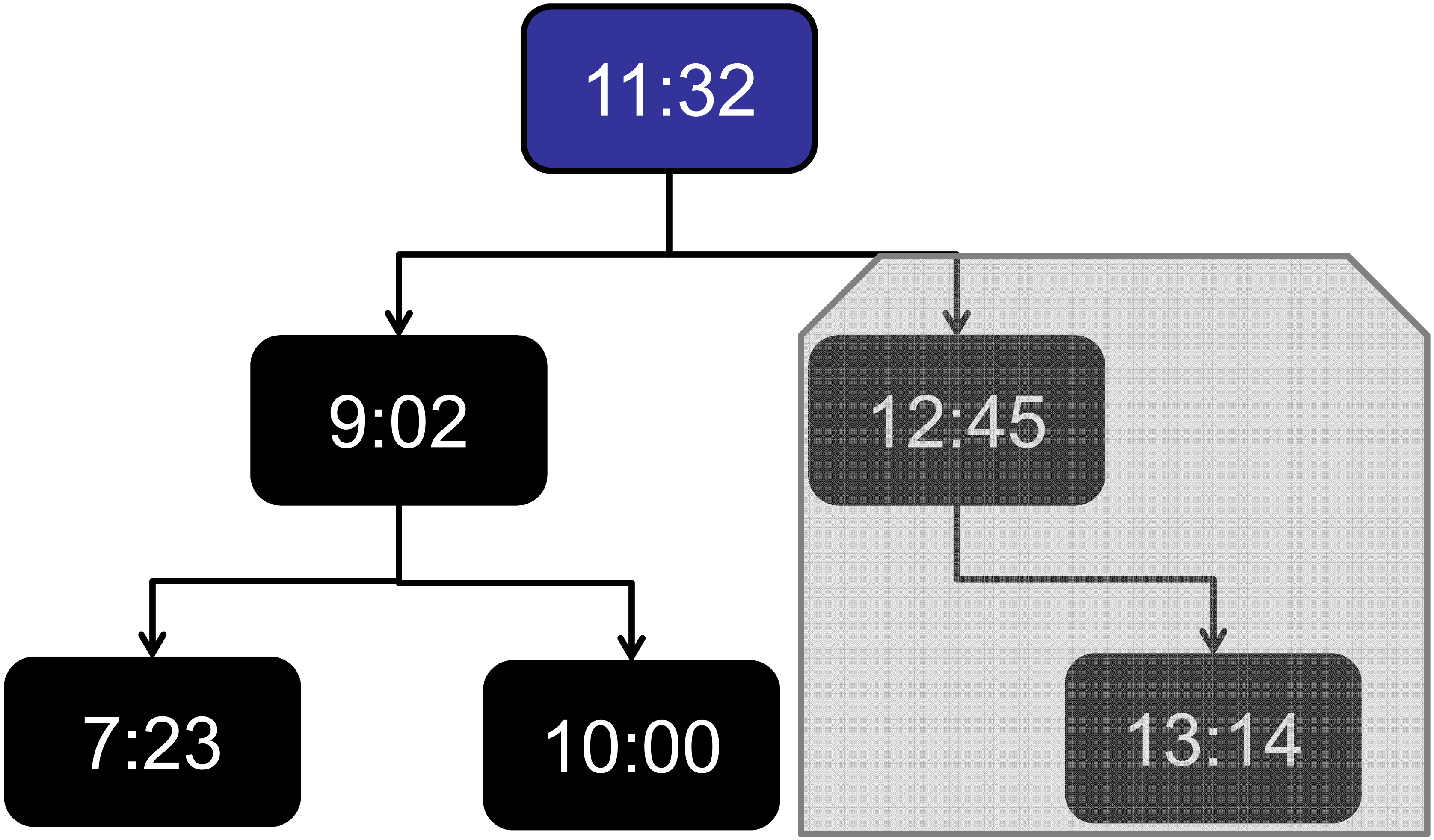
Binary Search Tree (BST)

BST Property: every key LEFT < key at node



Binary Search Tree (BST)

BST Property: every key RIGHT> key at node



Binary Search Tree (BST)

For every node x define:

- left[x] = left child
- right[x] = right child
- parent[x] = parent
- key[x] = value stored at node

BST Property:

- key[left[x]] < key[x] < key[right[x]]

Binary Search Tree (BST)

For every node x define:

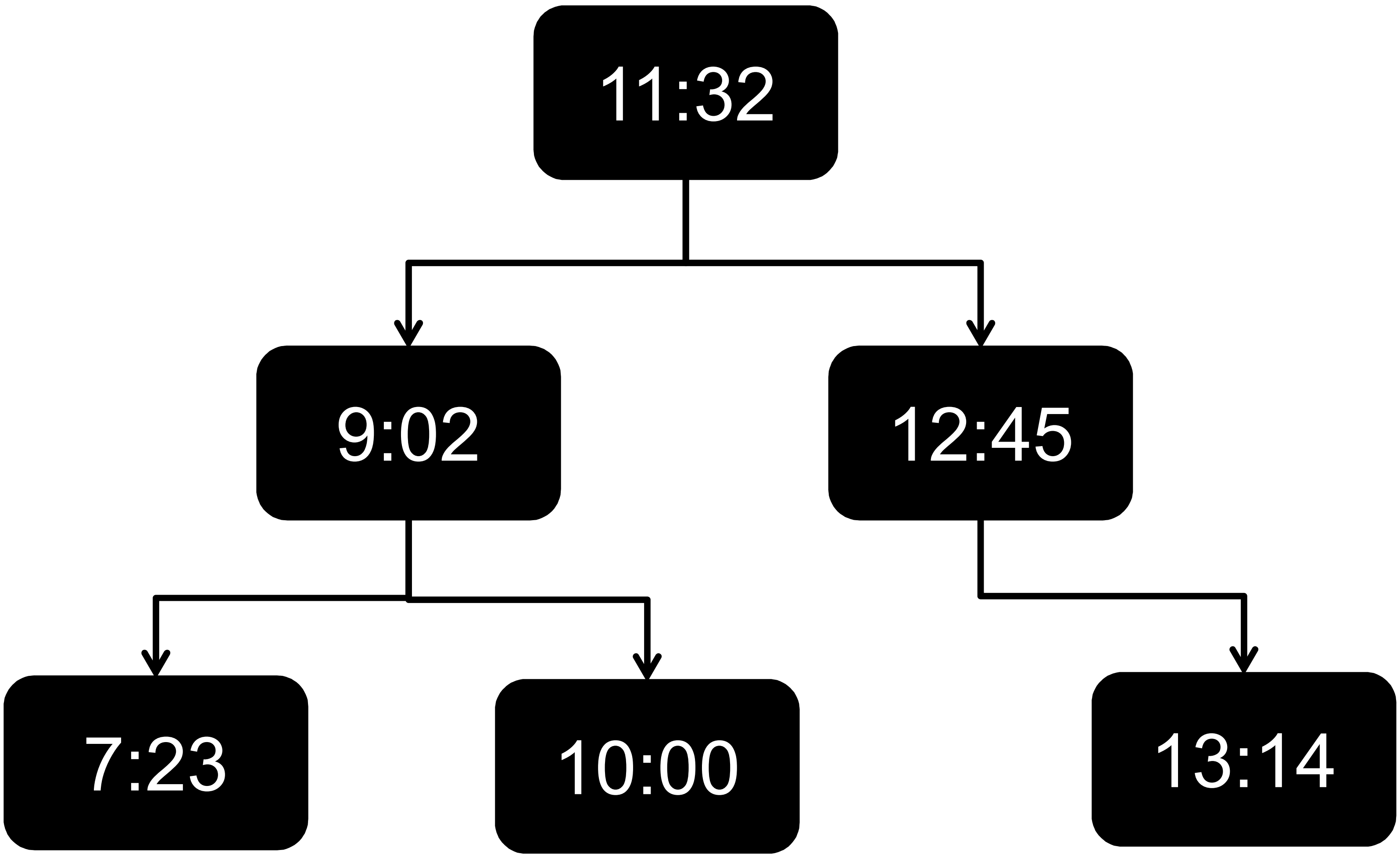
- left[x] = left child
- right[x] = right child
- parent[x] = parent
- key[x] = value stored at node

BST Property:

- key[left[x]] < key[x] < key[right[x]]

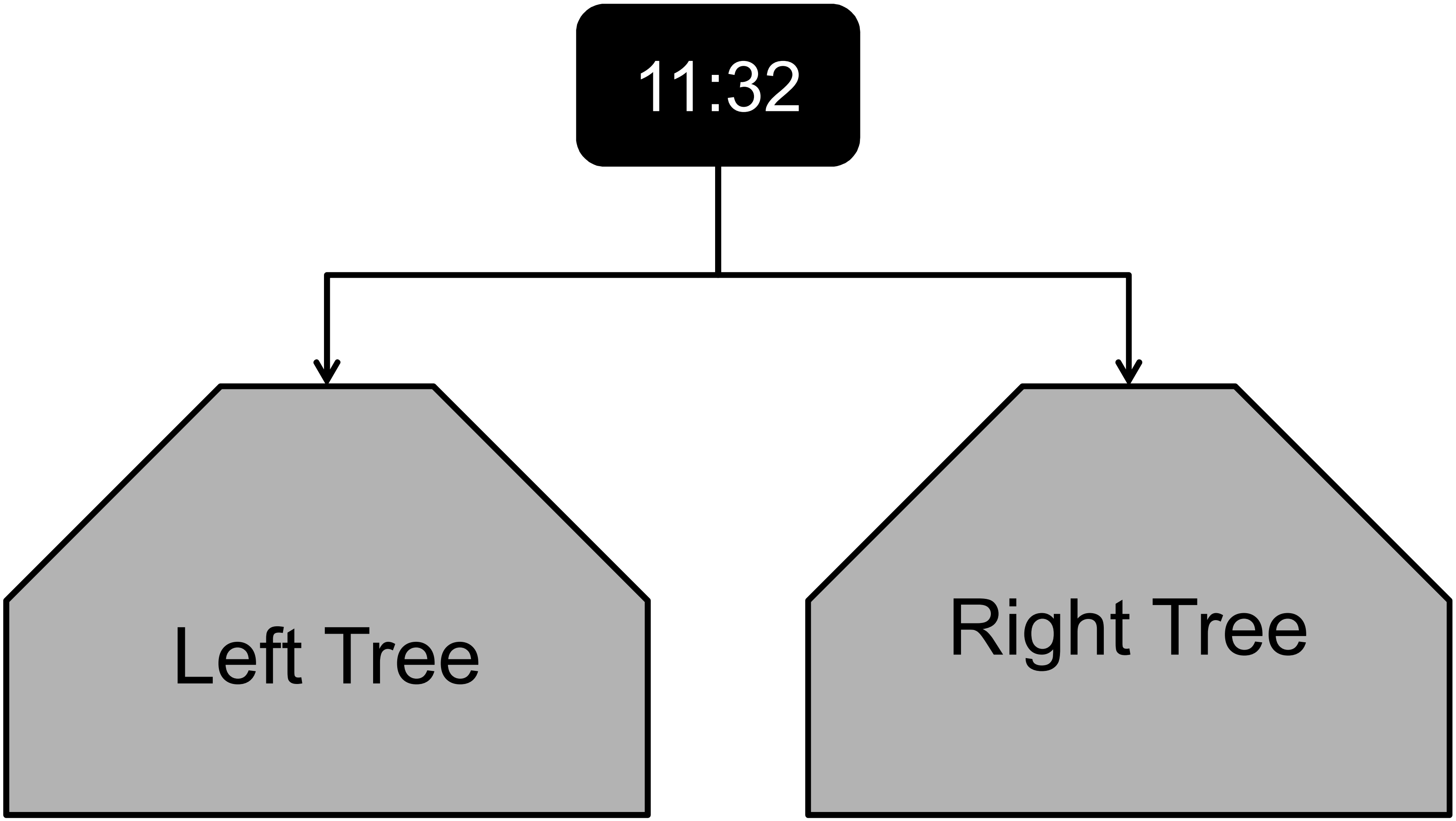
Assume keys are unique!

Binary Search Tree



Binary Search Tree (BST)

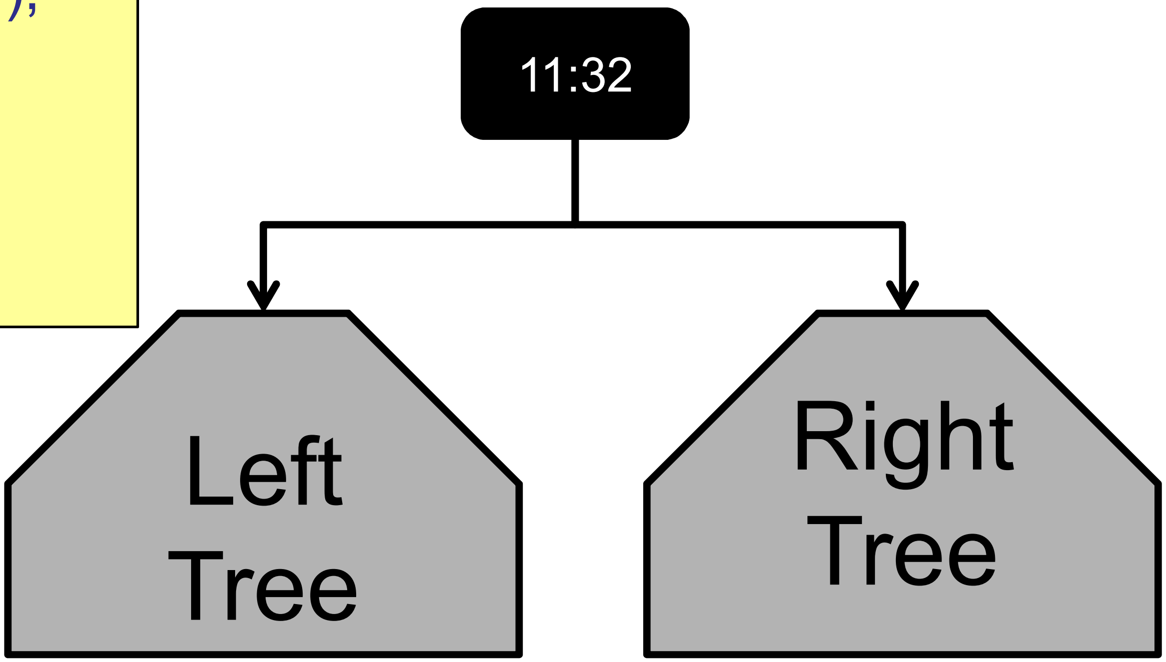
Recursive definition:



Binary Search Tree (BST)

Recursive search algorithm:

```
search(int v){
   if (key==v){
        return true;
   else if (key < v){
        return right.search(v);
   else if (key > v){
        return left.search(v);
}
```

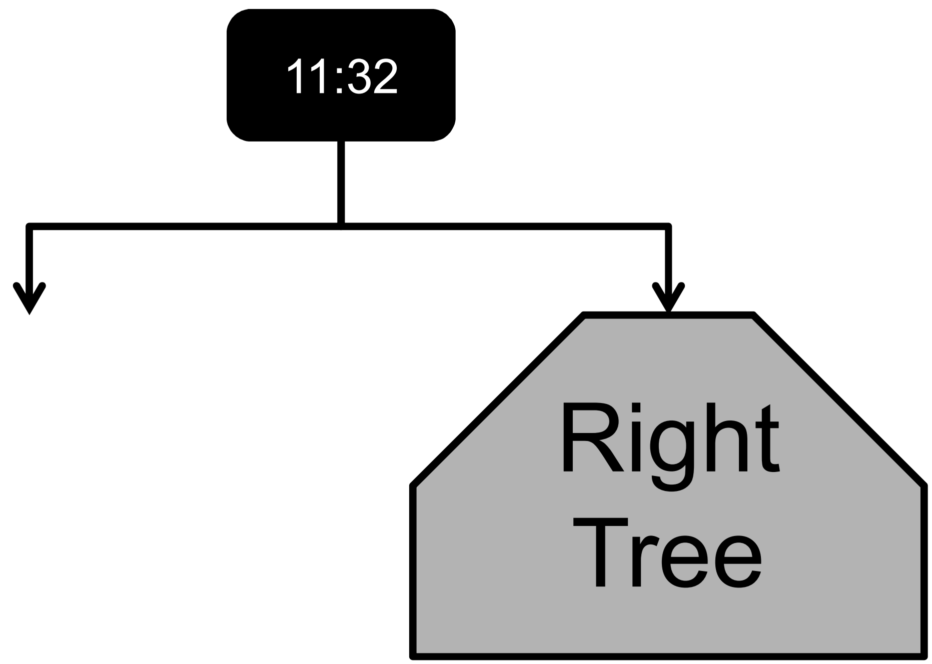


Binary Search Tree (BST)

Recursive search algorithm:

```
search(int v){
    if (key==v){
        return true;
    else if (key < v){
        return right.search(v);
    else if (key > v){
        return left.search(v);
}
```

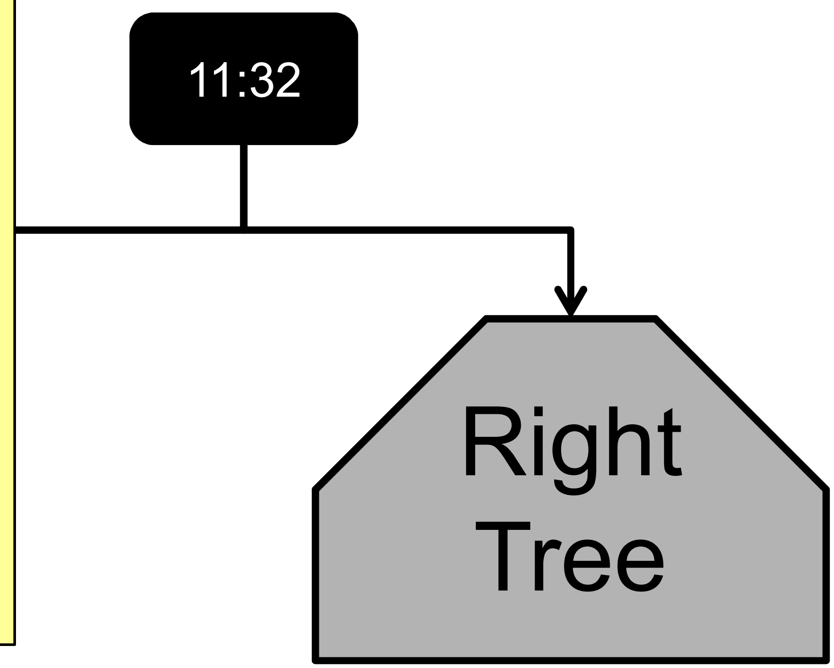
What if there is no left or right sub-tree?



Binary Search Tree (BST)

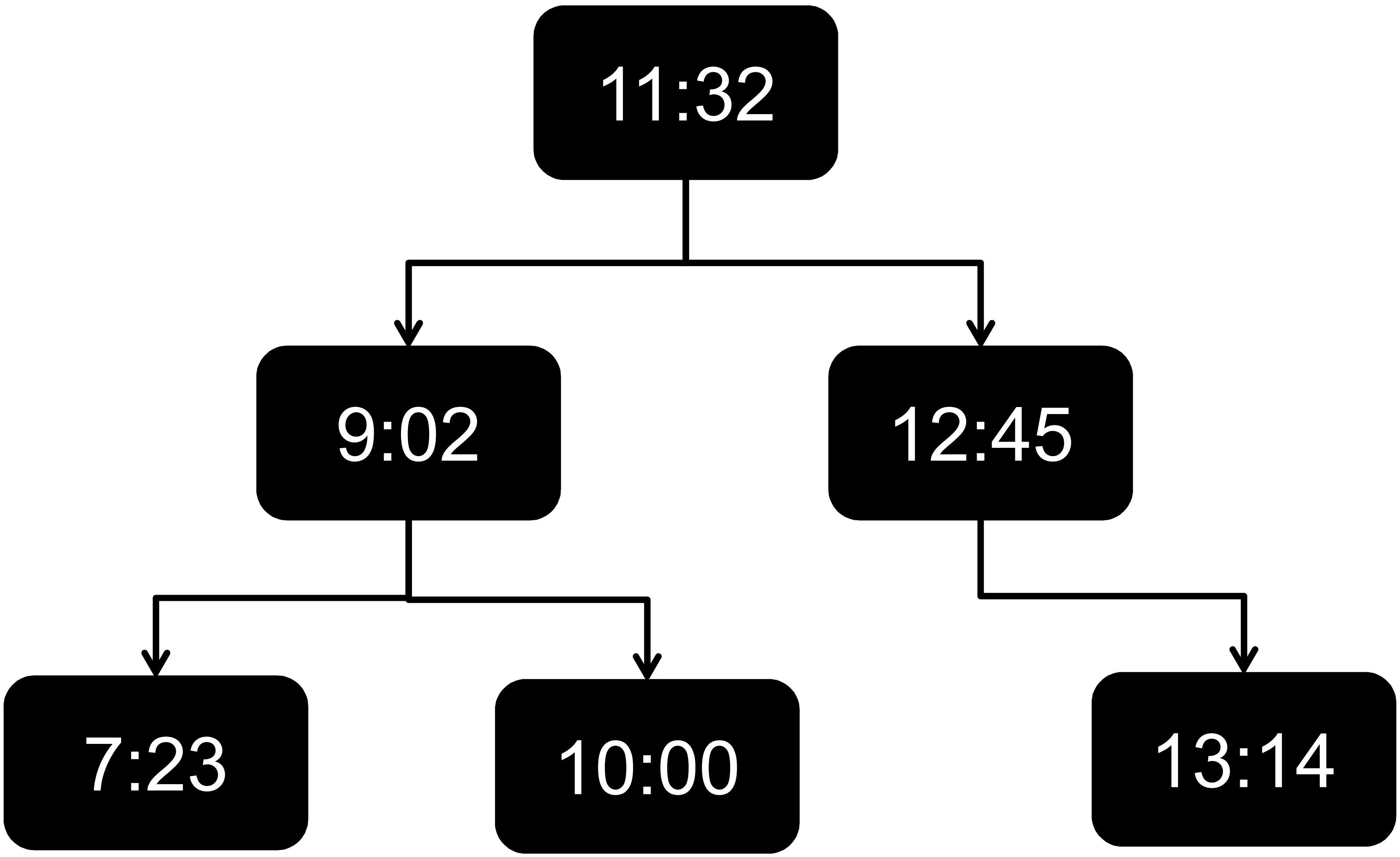
Recursive search algorithm:

```
search(int v){
    if (key==v) return true;
    else if (key < v){
        if (left == null) return false;
        else return right.search(v);
    }
    else if (key > v){
        if (right == null) return false;
        else return left.search(v);
    }
}
```



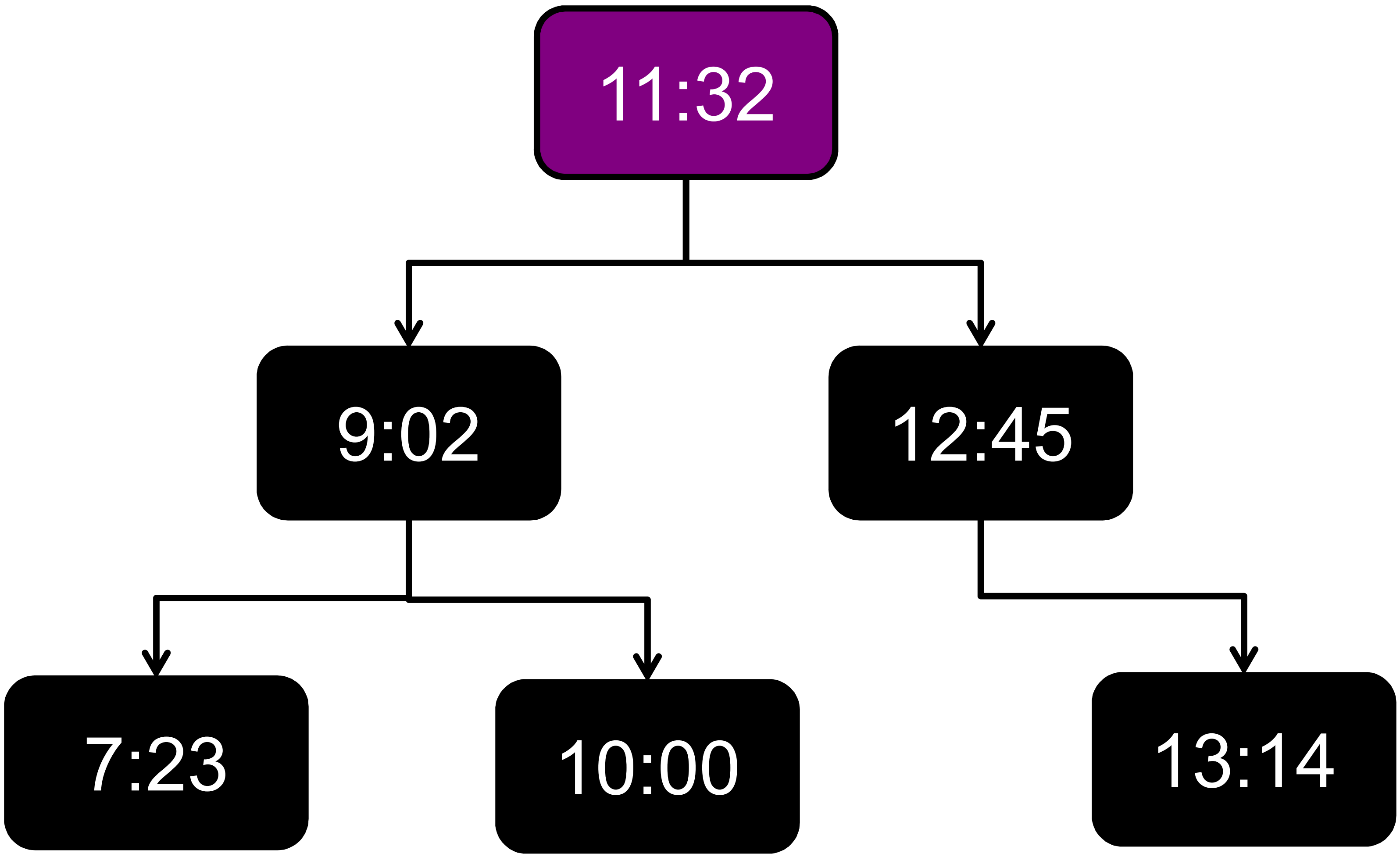
Binary Search Tree

Search for 12:52:



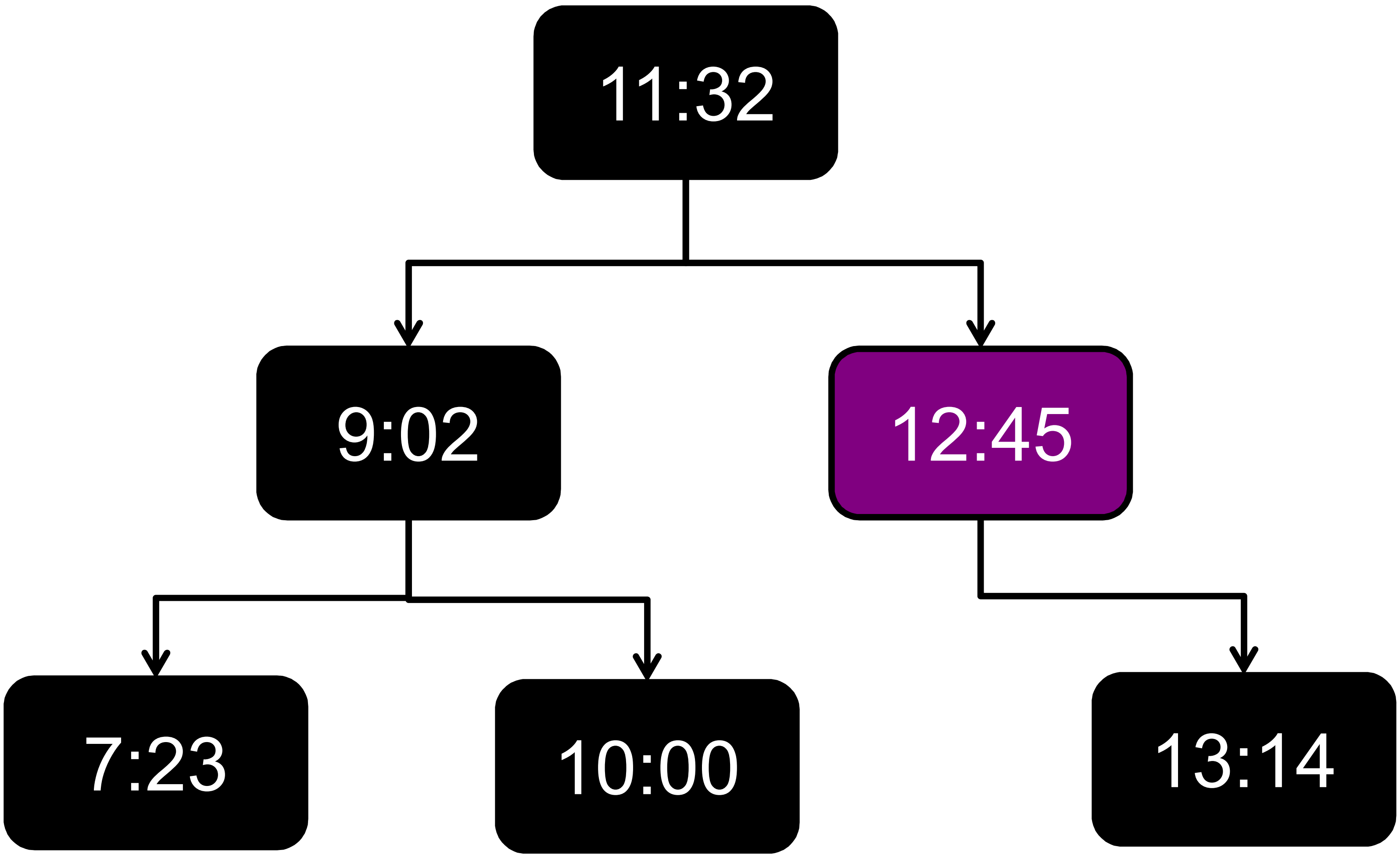
Binary Search Tree

Search for 12:52:



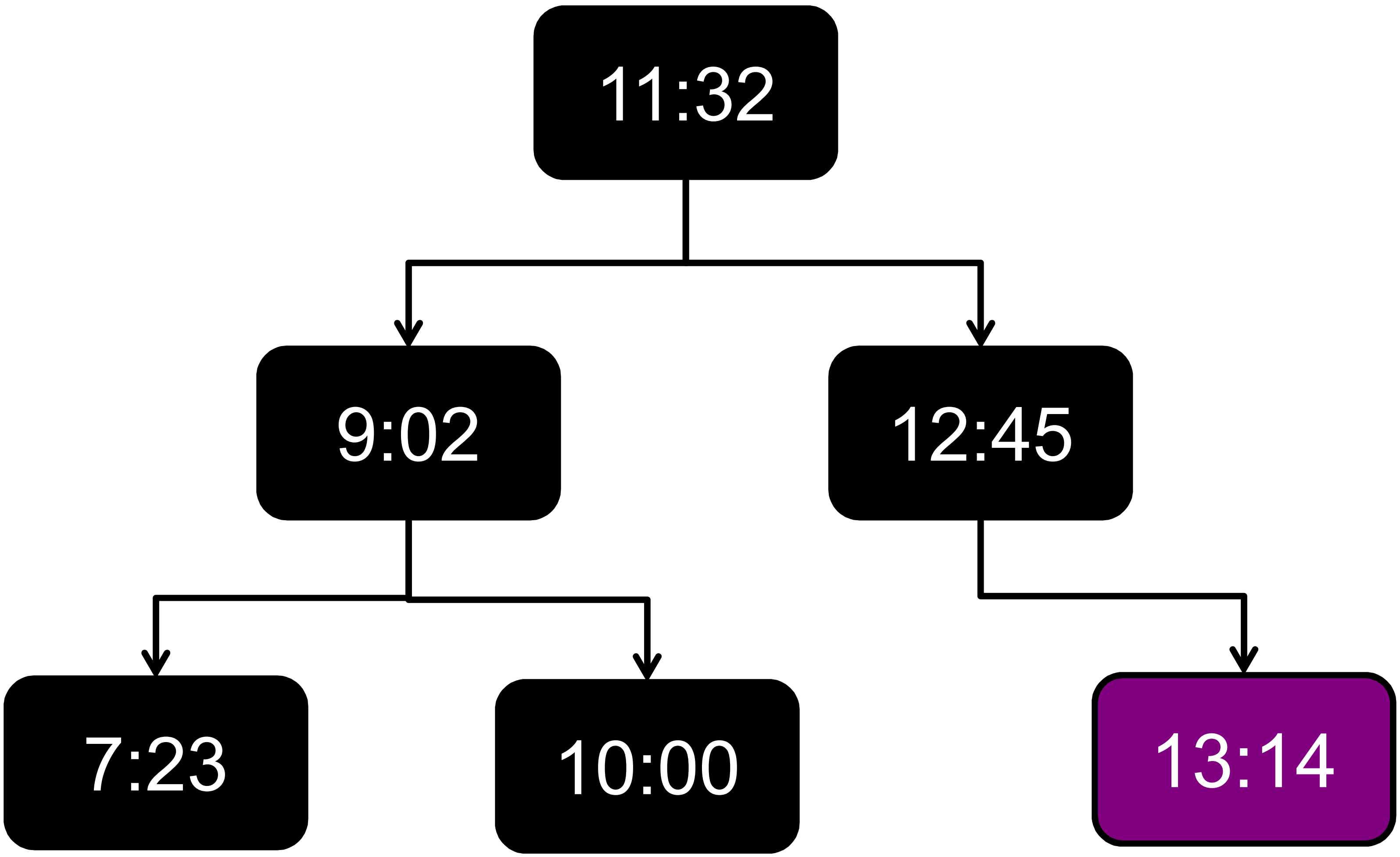
Binary Search Tree

Search for 12:52:



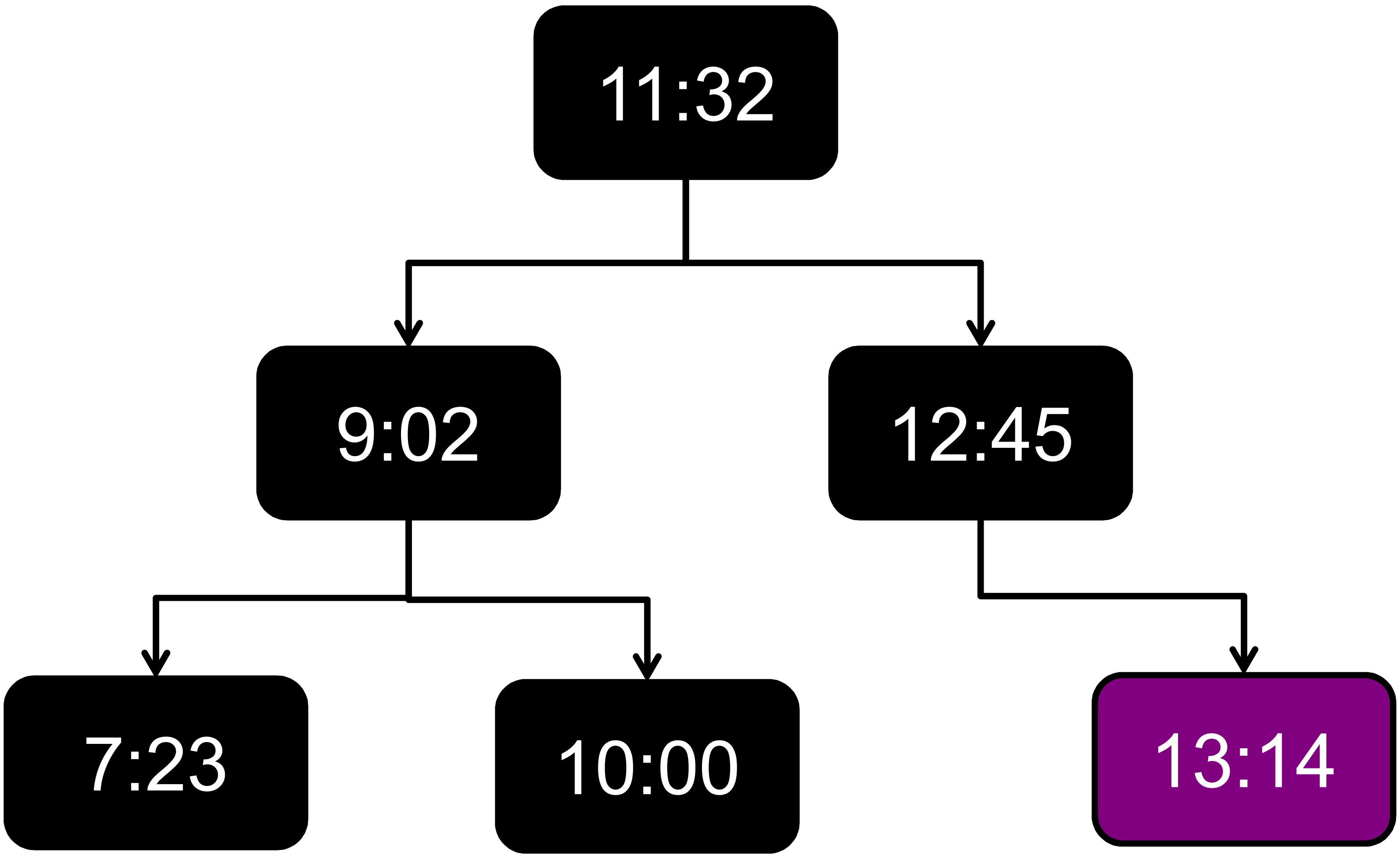
Binary Search Tree

Search for 12:52:



Binary Search Tree

Search for 12:52: return: FALSE



Binary Search Tree (BST)

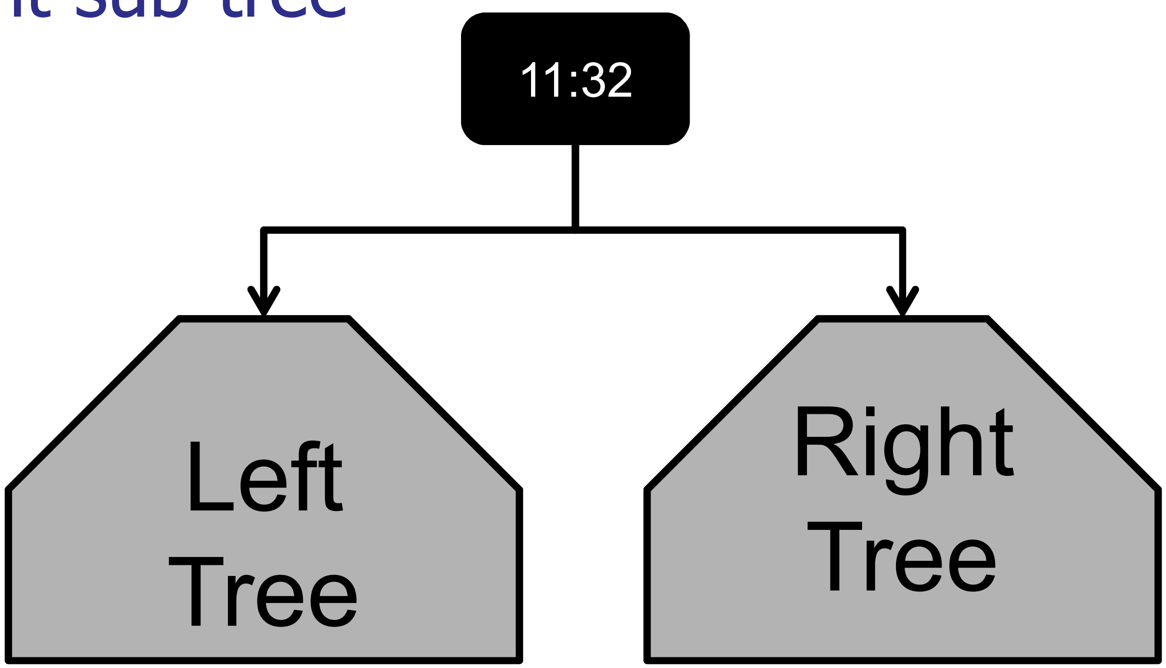
Insert value v:

if (v < key)

insert key in left sub-tree

else if (v > key)

insert key in right sub-tree

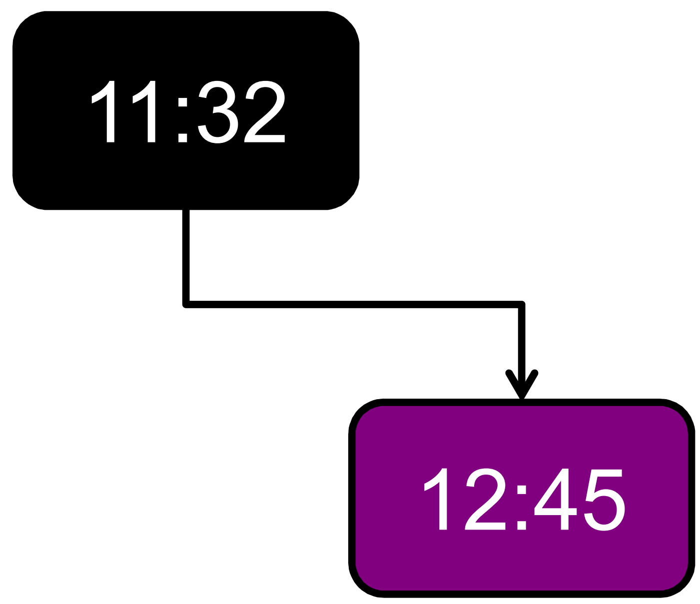


Binary Search Tree

insert 11:32

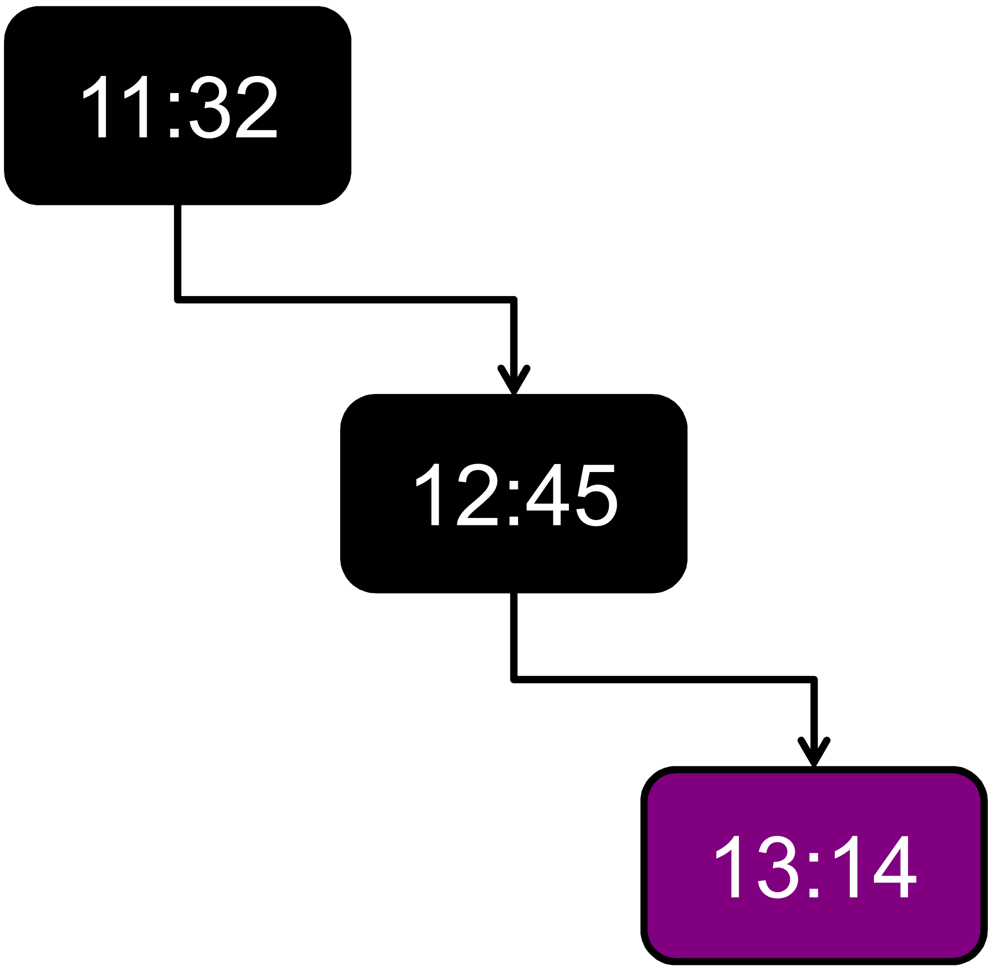
Binary Search Tree

insert 12:45



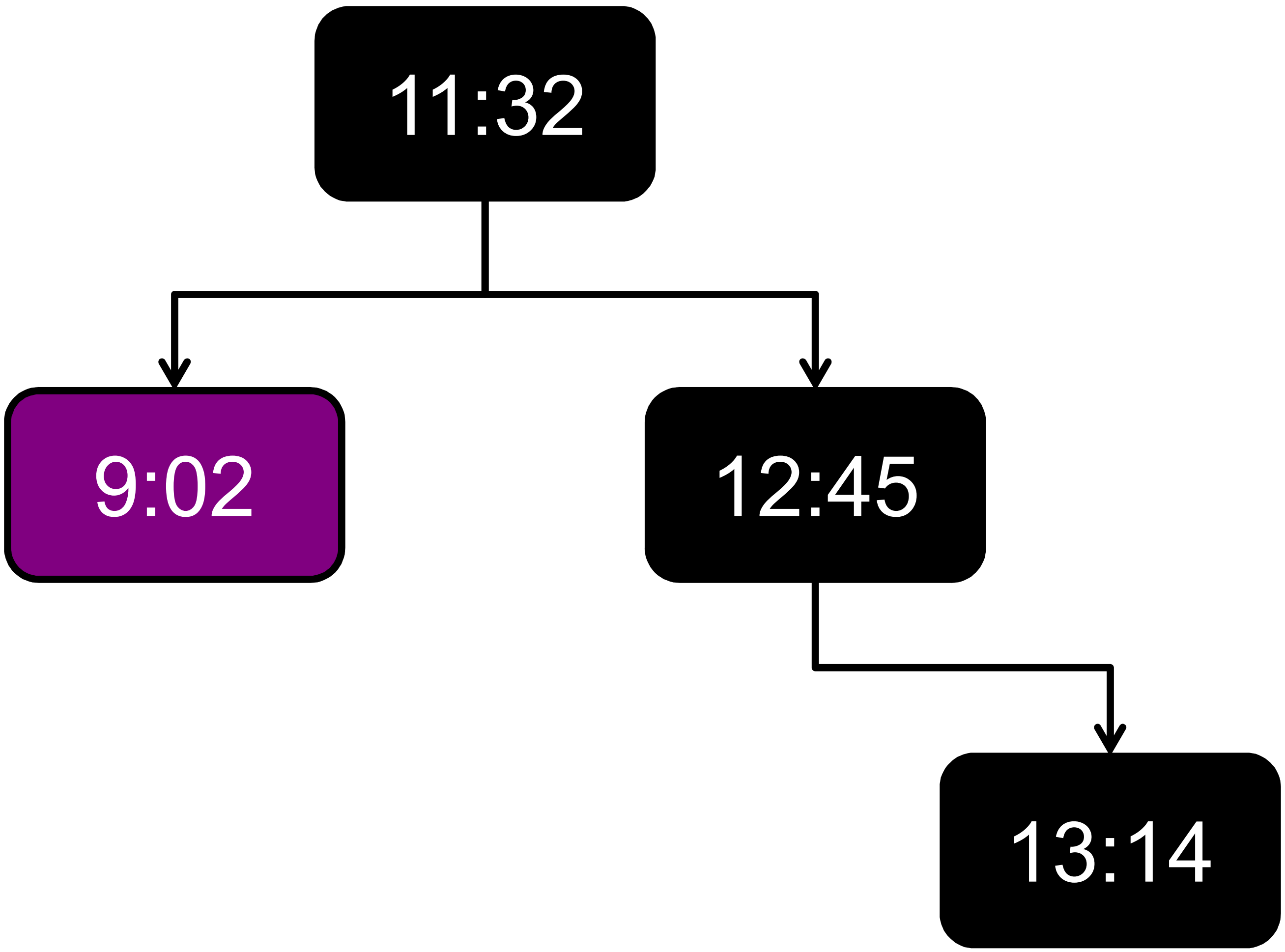
Binary Search Tree

insert 13:14



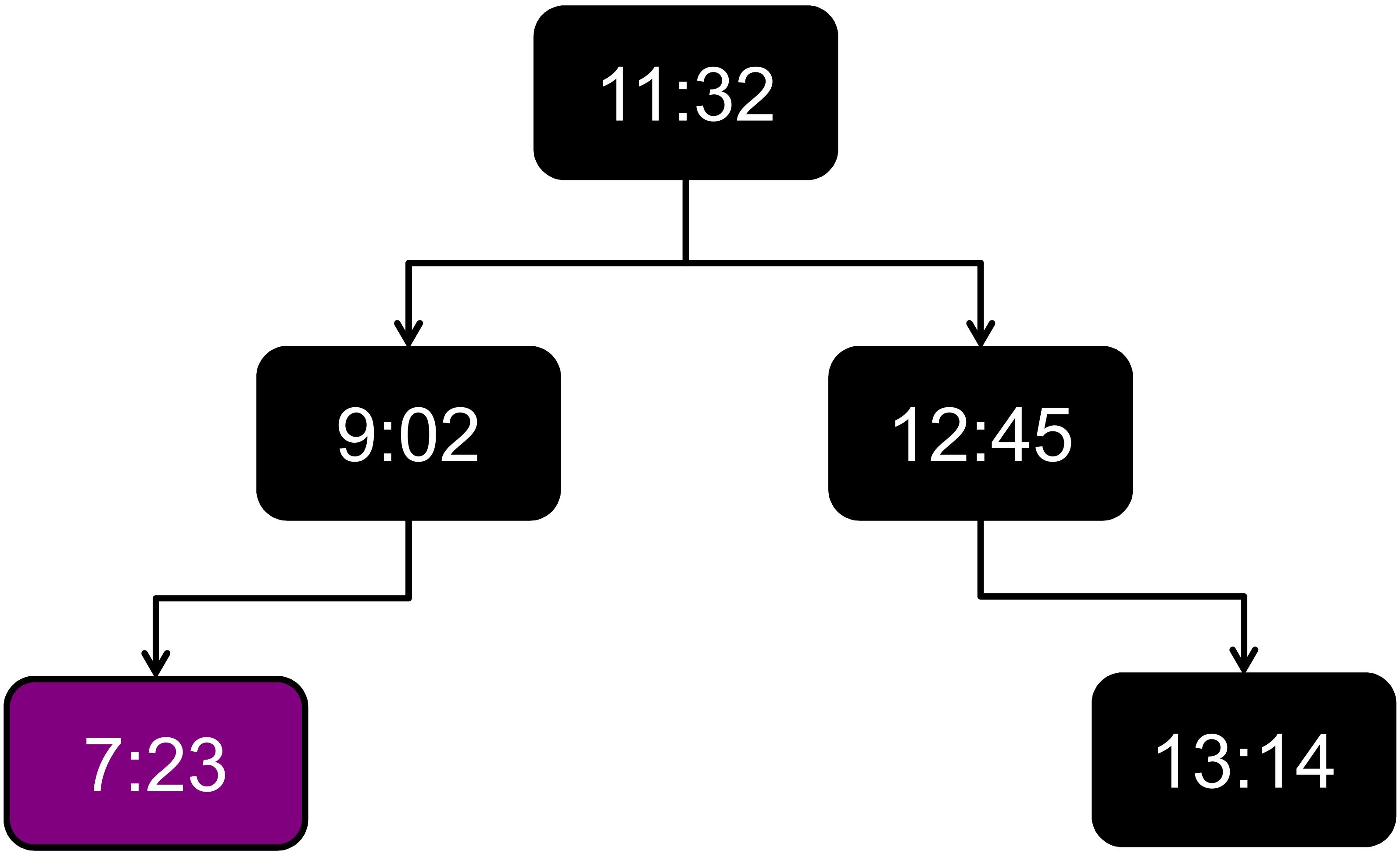
Binary Search Tree

insert 9:02



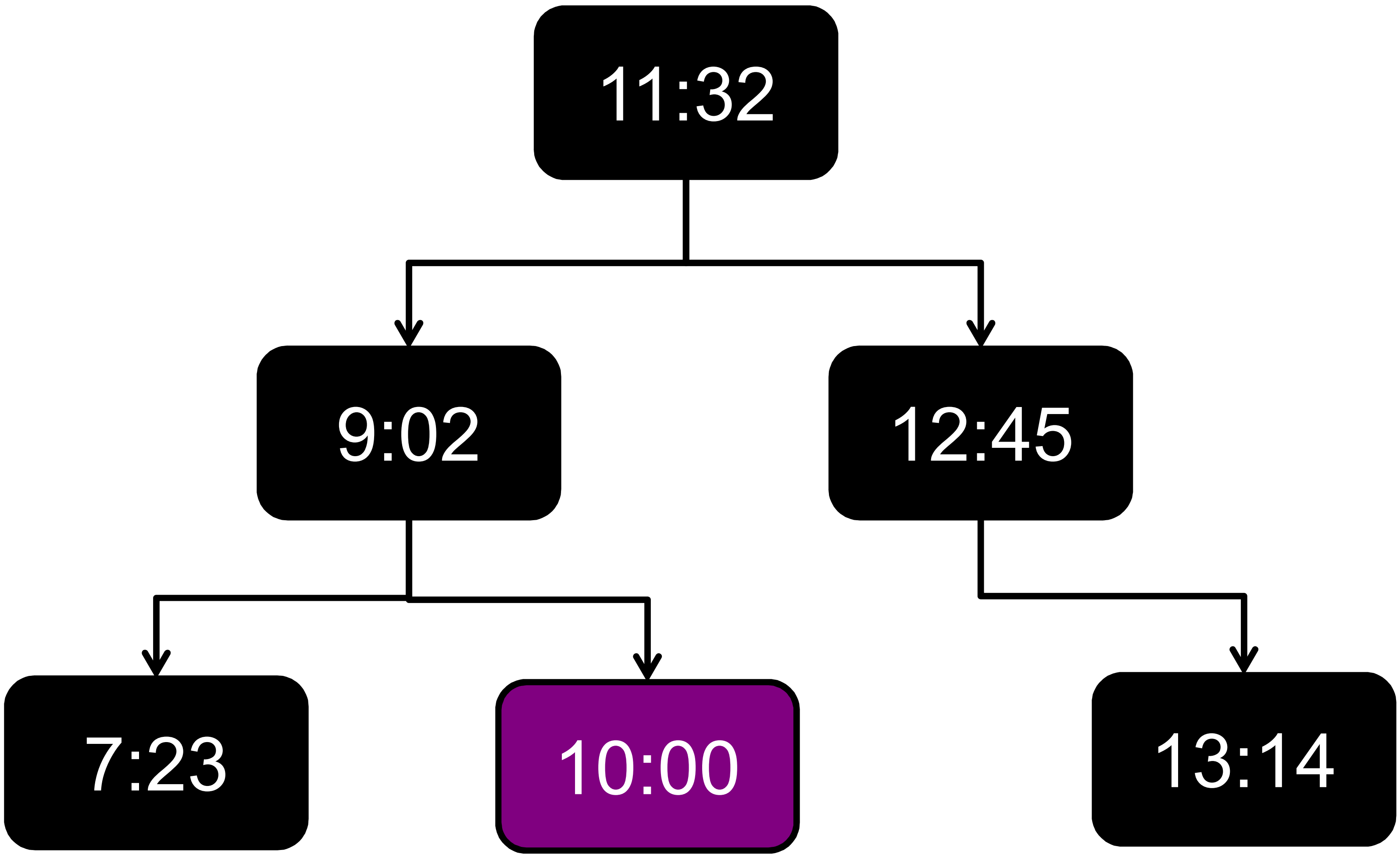
Binary Search Tree

insert 7:23



Binary Search Tree

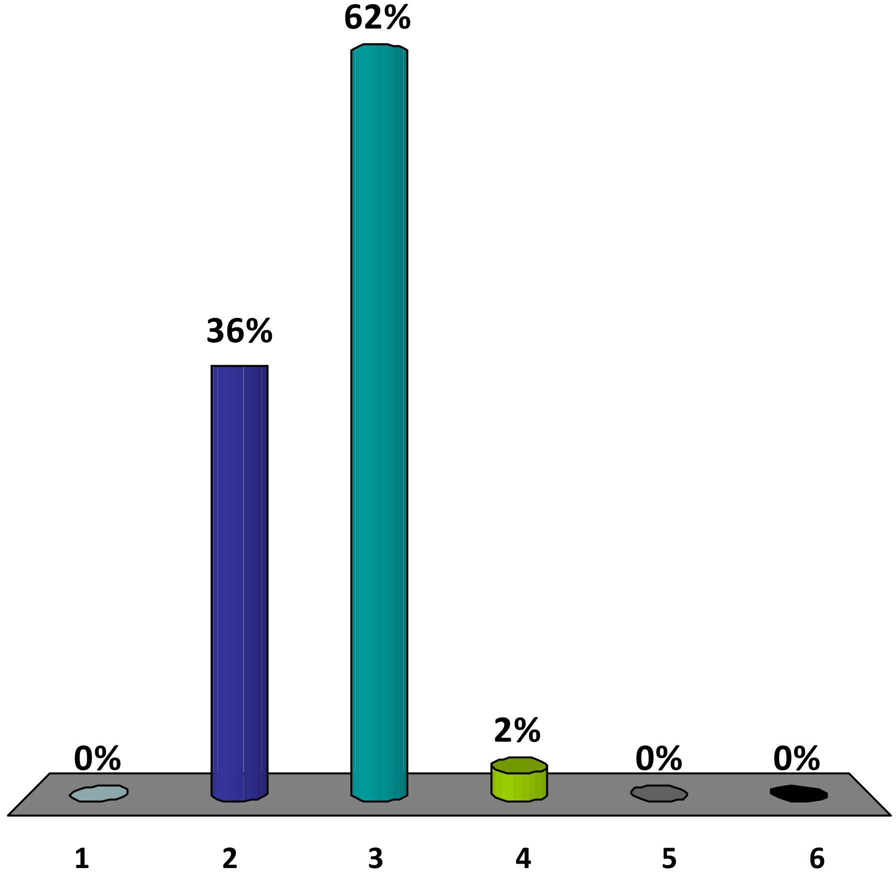
insert 10:00



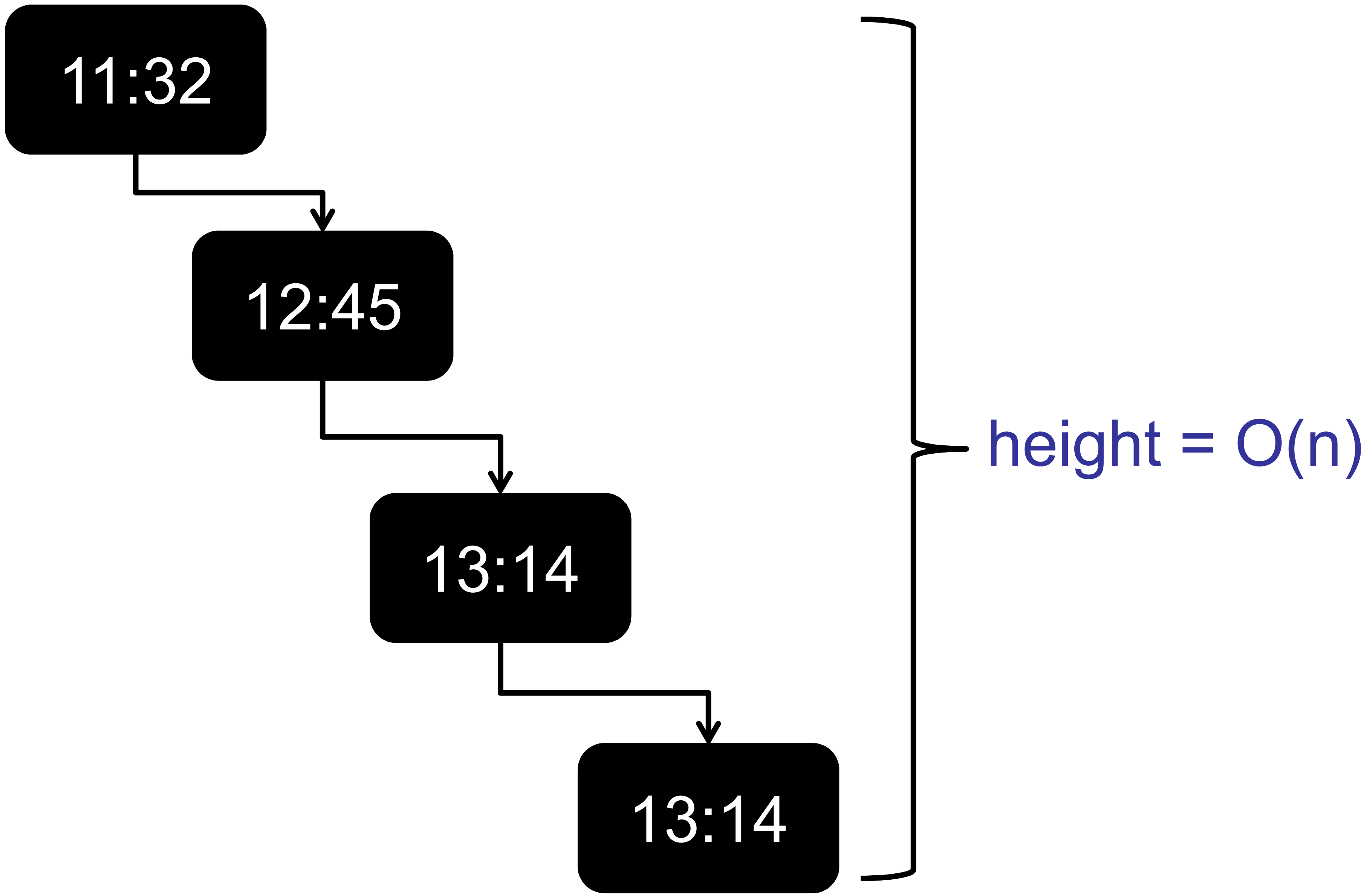
Binary Search Tree

What is the worst-case running time of search in a BST?

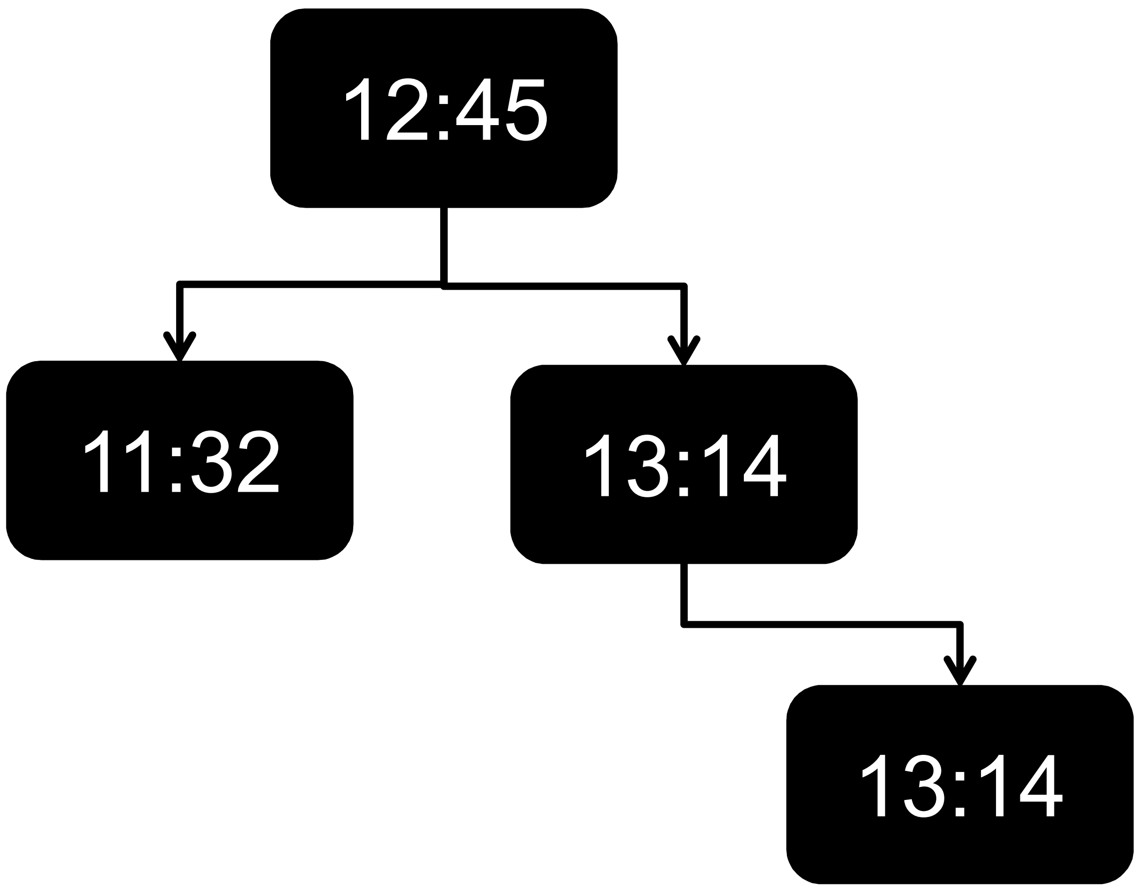
1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(n^3)$
6. $O(2^n)$



Binary Search Tree



Binary Search Tree



balanced = minimal height

Binary Search Tree

Summary:

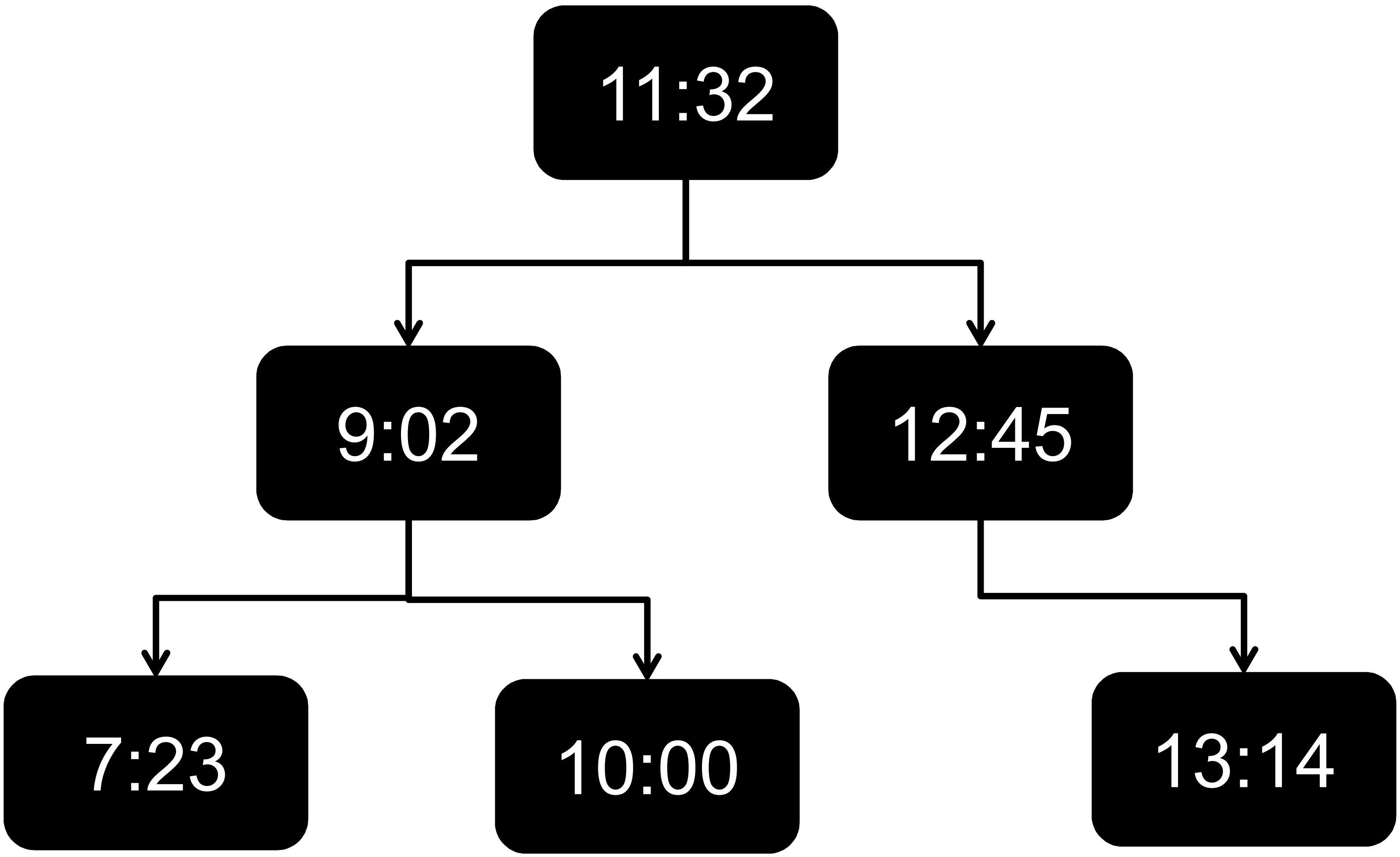
- search: O(h)
- insert: O(h)

Other operations: O(h)

- findMax
- findMin
- predecessor
- successor
- delete

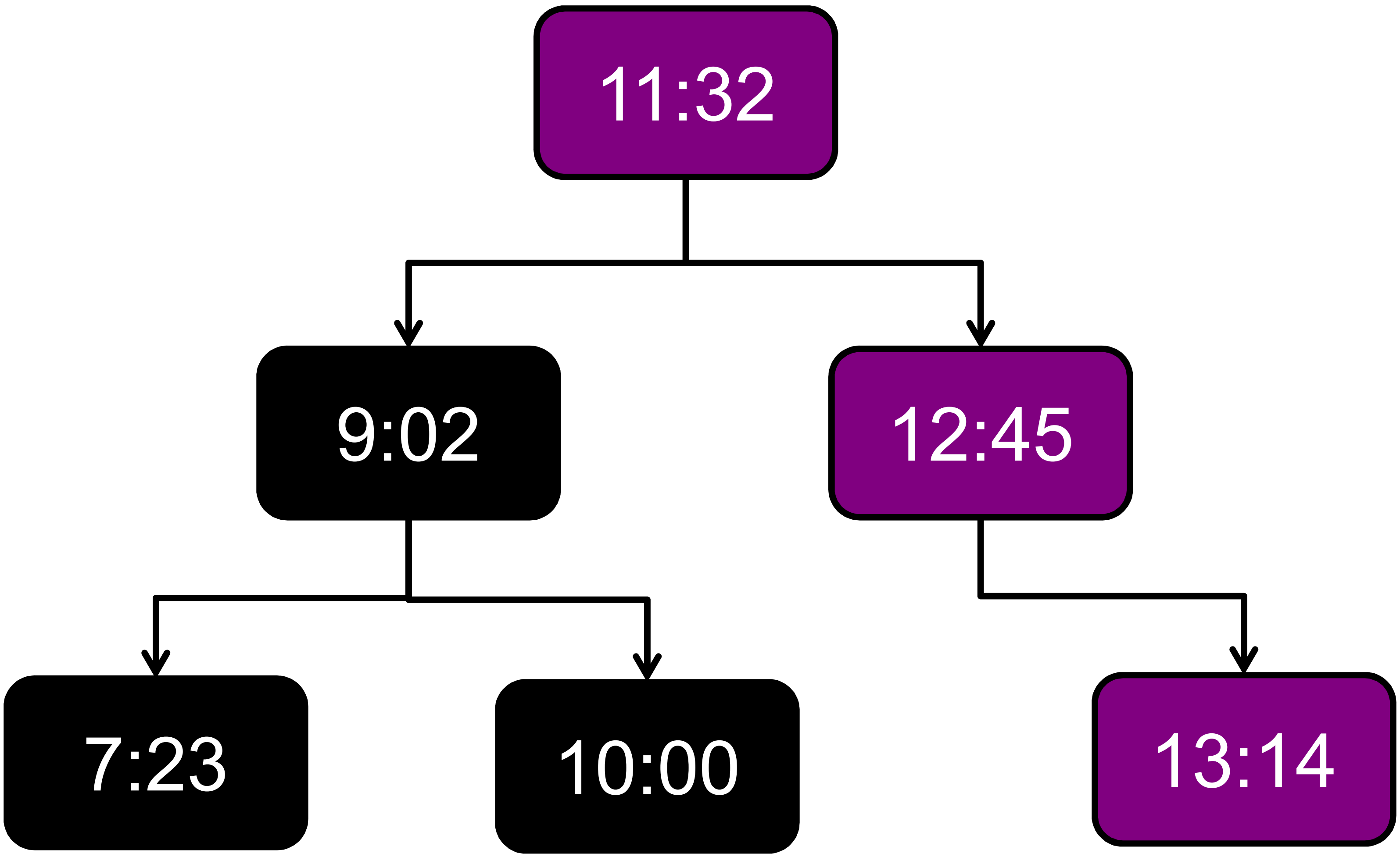
Binary Search Tree

findMax()



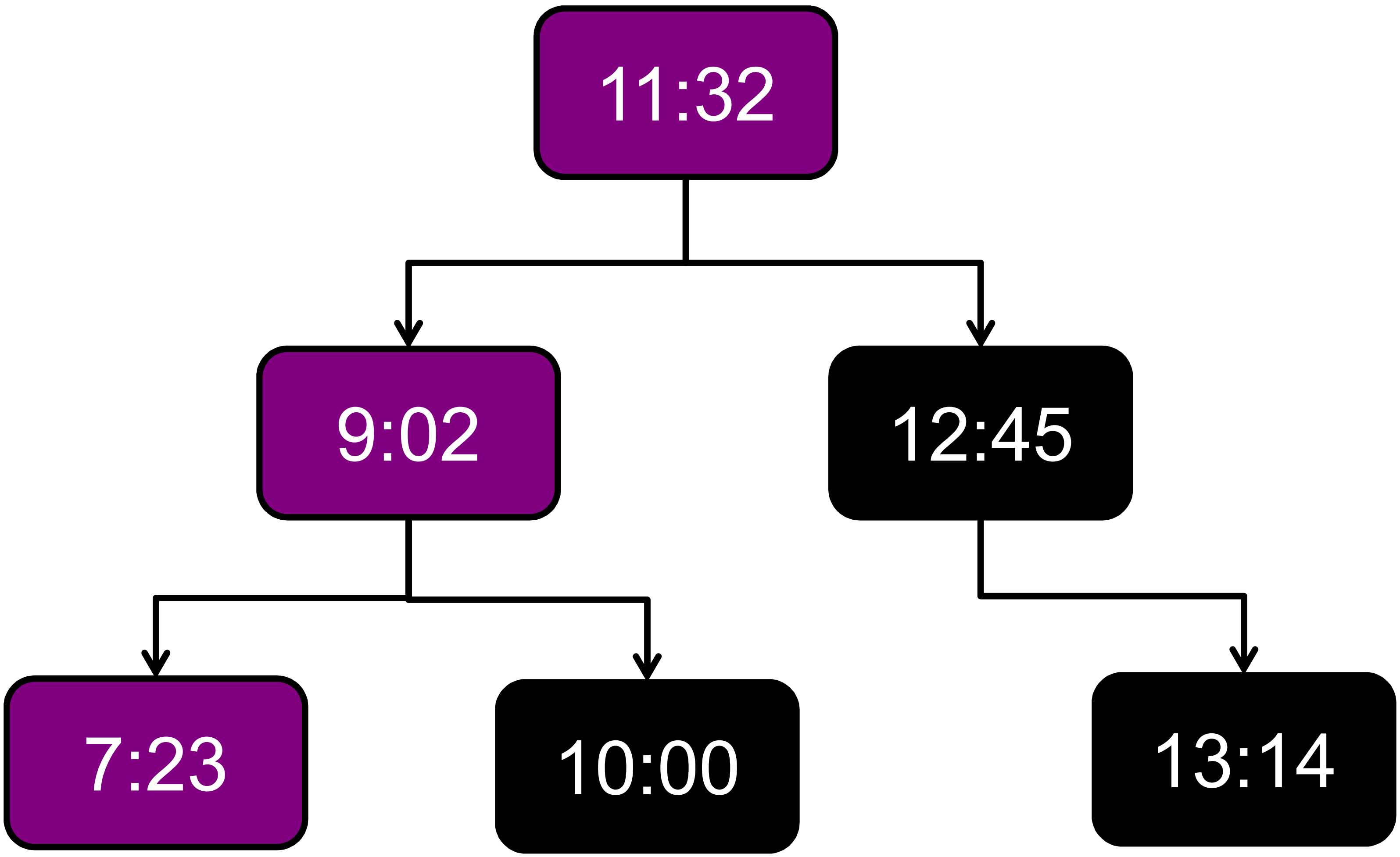
Binary Search Tree

findMax()



Binary Search Tree

findMin()



Binary Search Tree

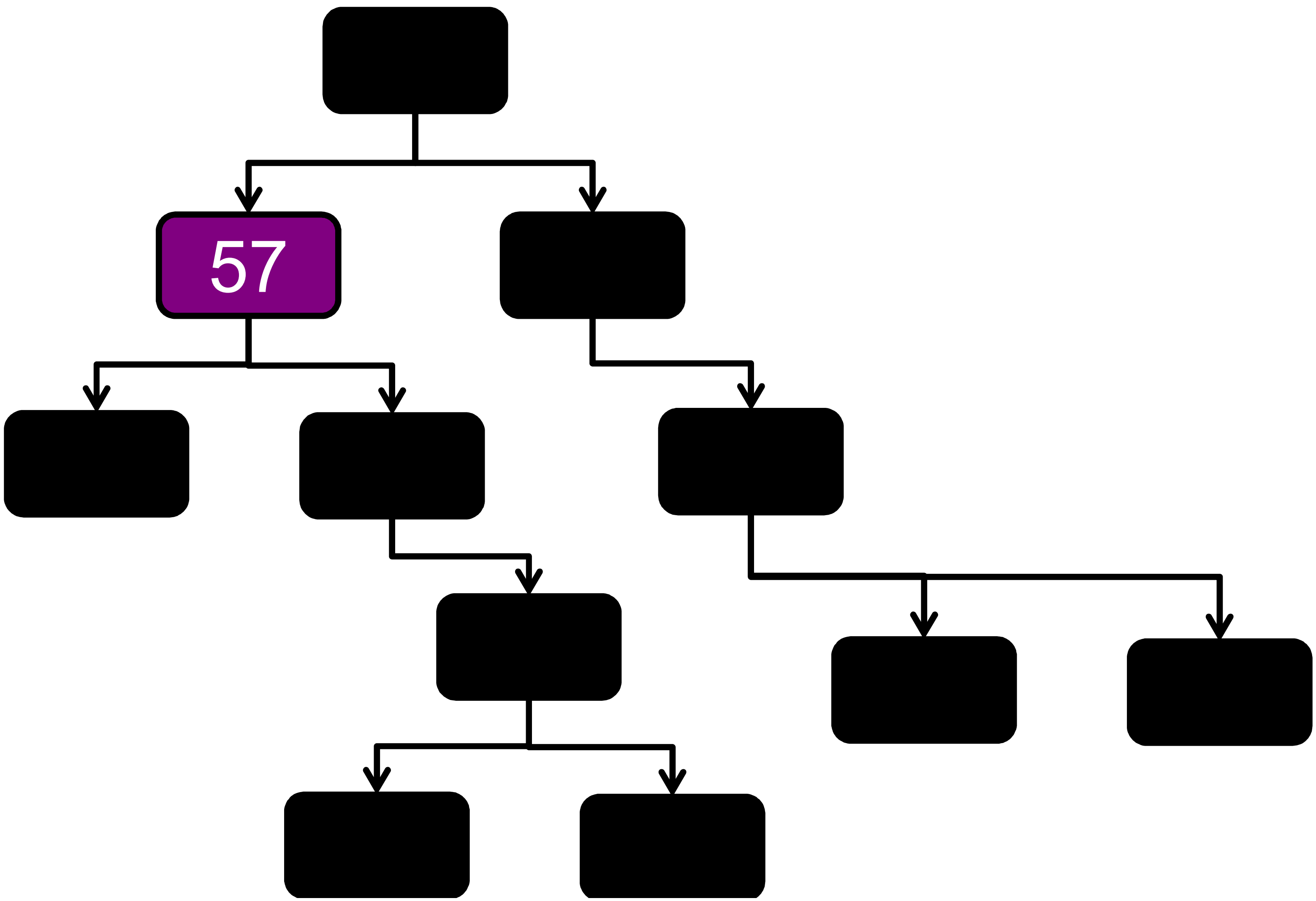
```
findMax(){
   if (right==null) {
      return key;
   }
   else{
      return right.findMax();
   }
```

Time: O(h)

```
findMin(){
   if (left==null) {
      return key;
   }
   else{
      return left.findMin();
   }
```

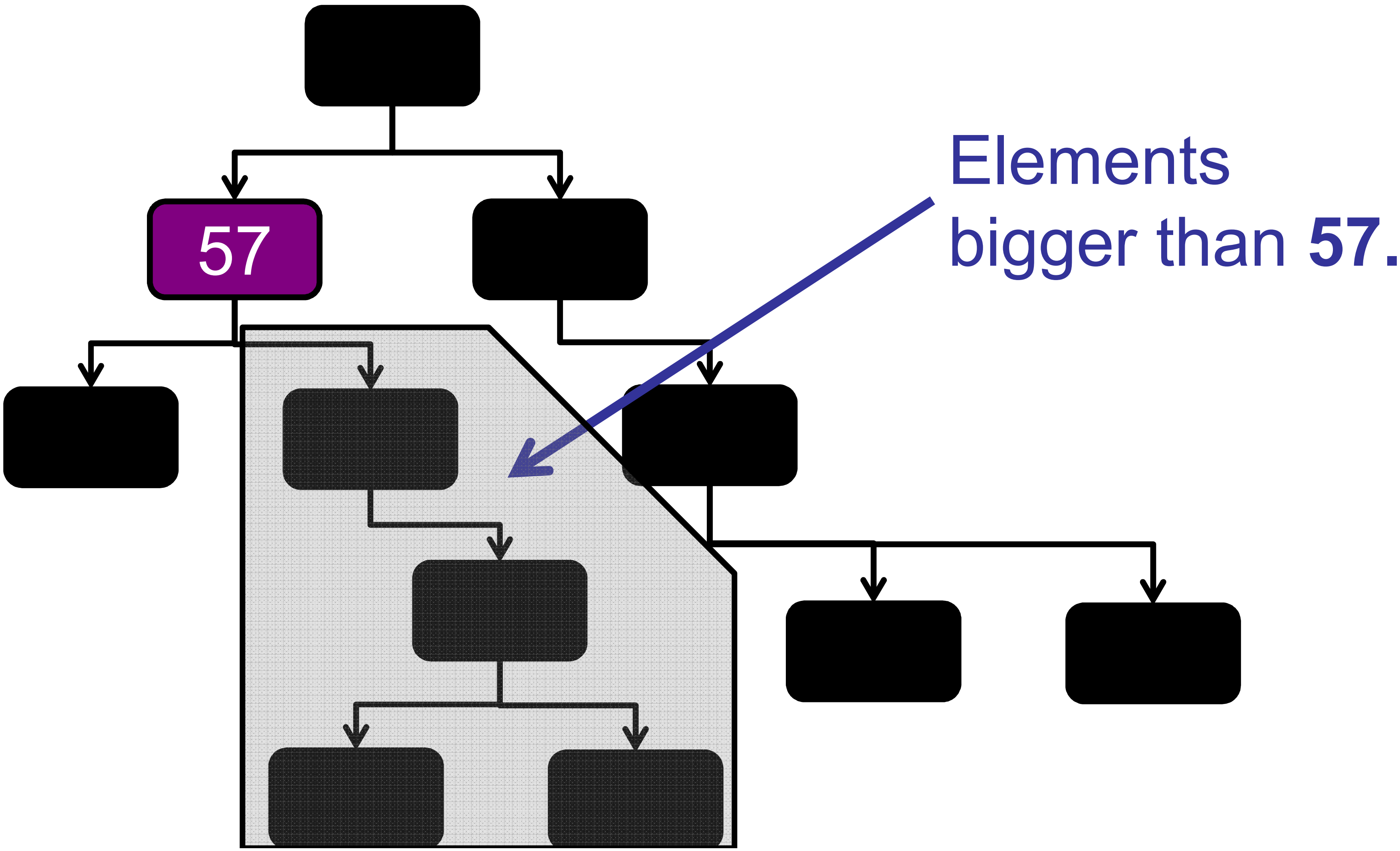
Binary Search Tree

successor()



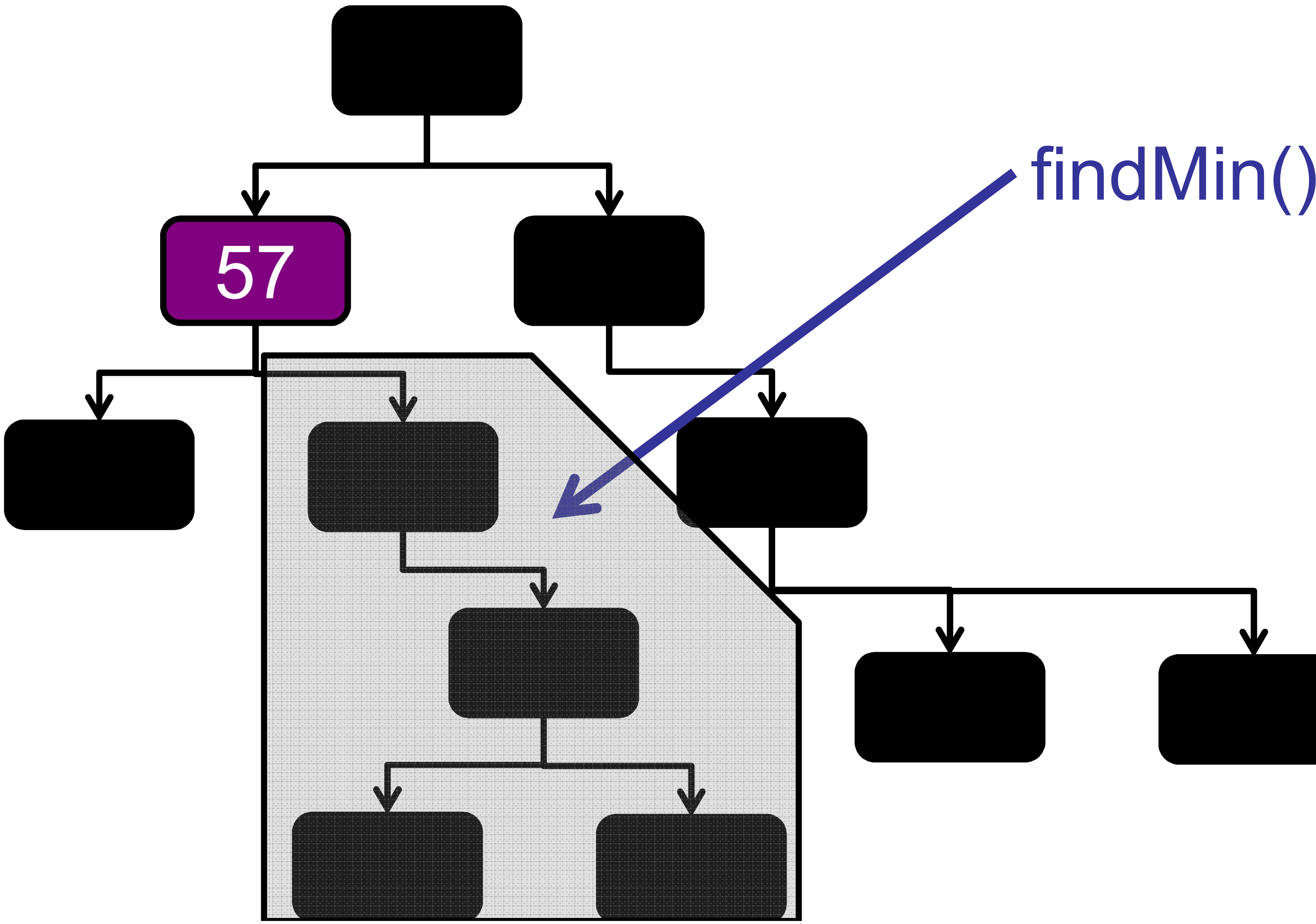
Binary Search Tree

successor()



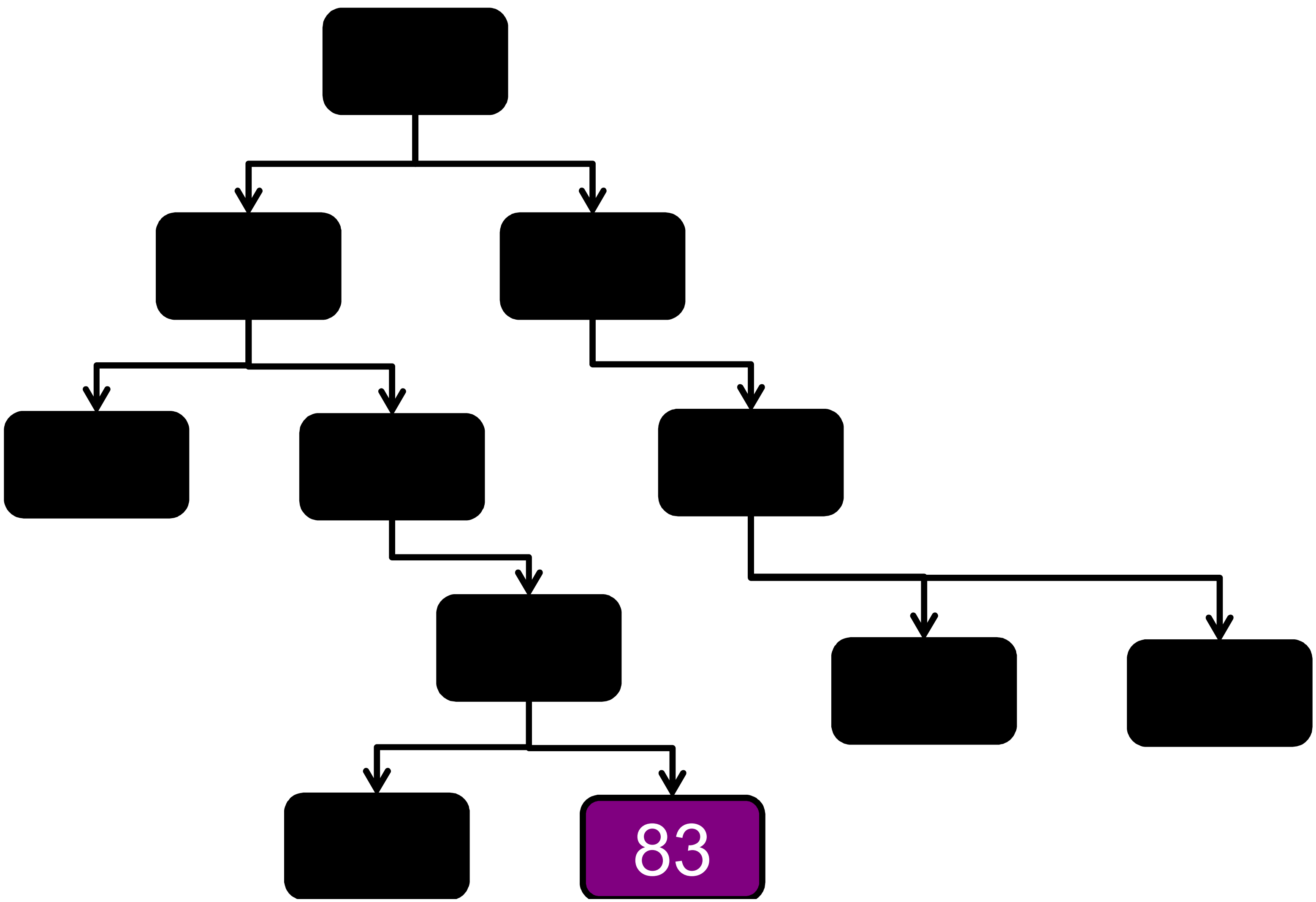
Binary Search Tree

successor()



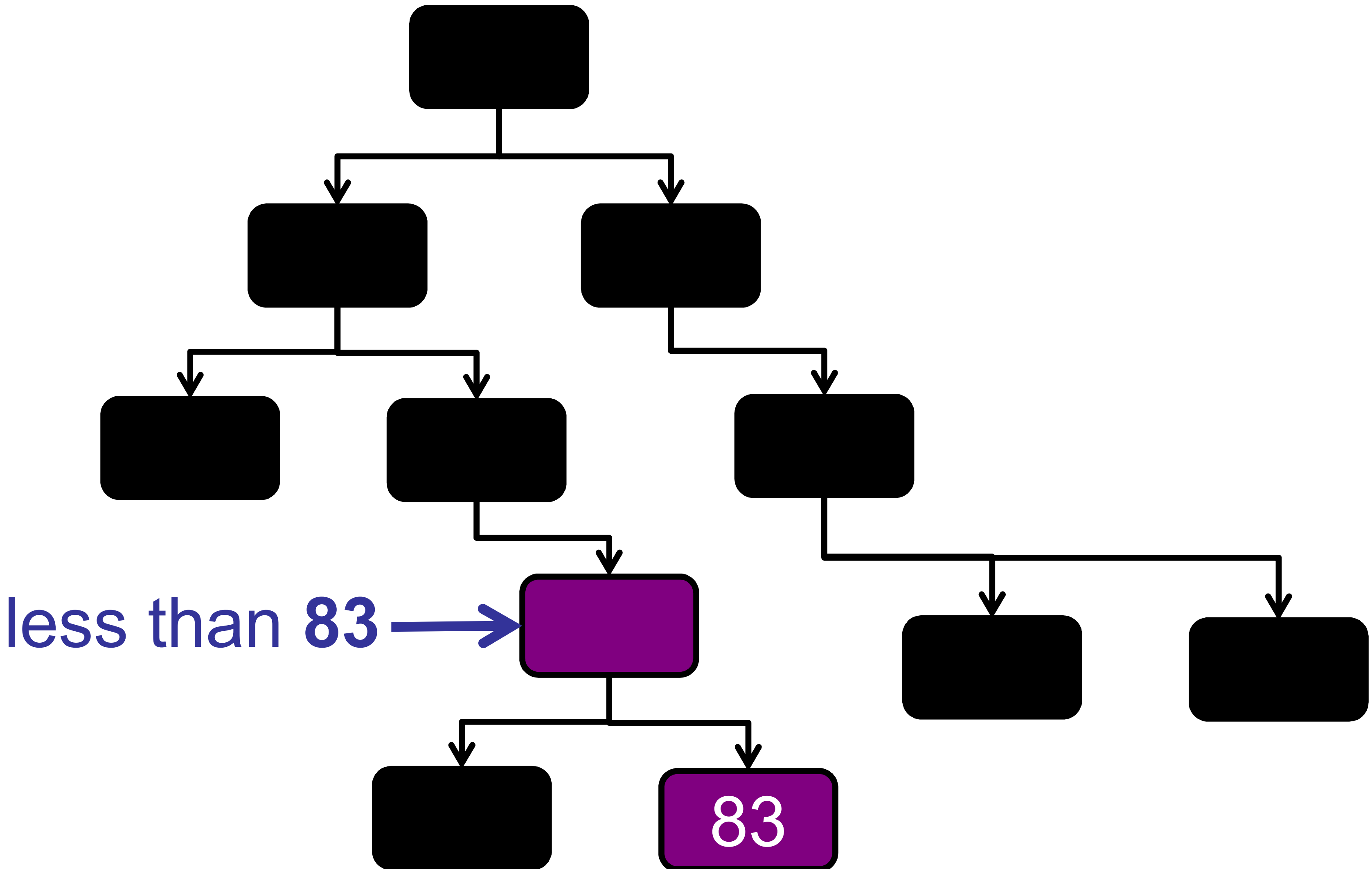
Binary Search Tree

successor()



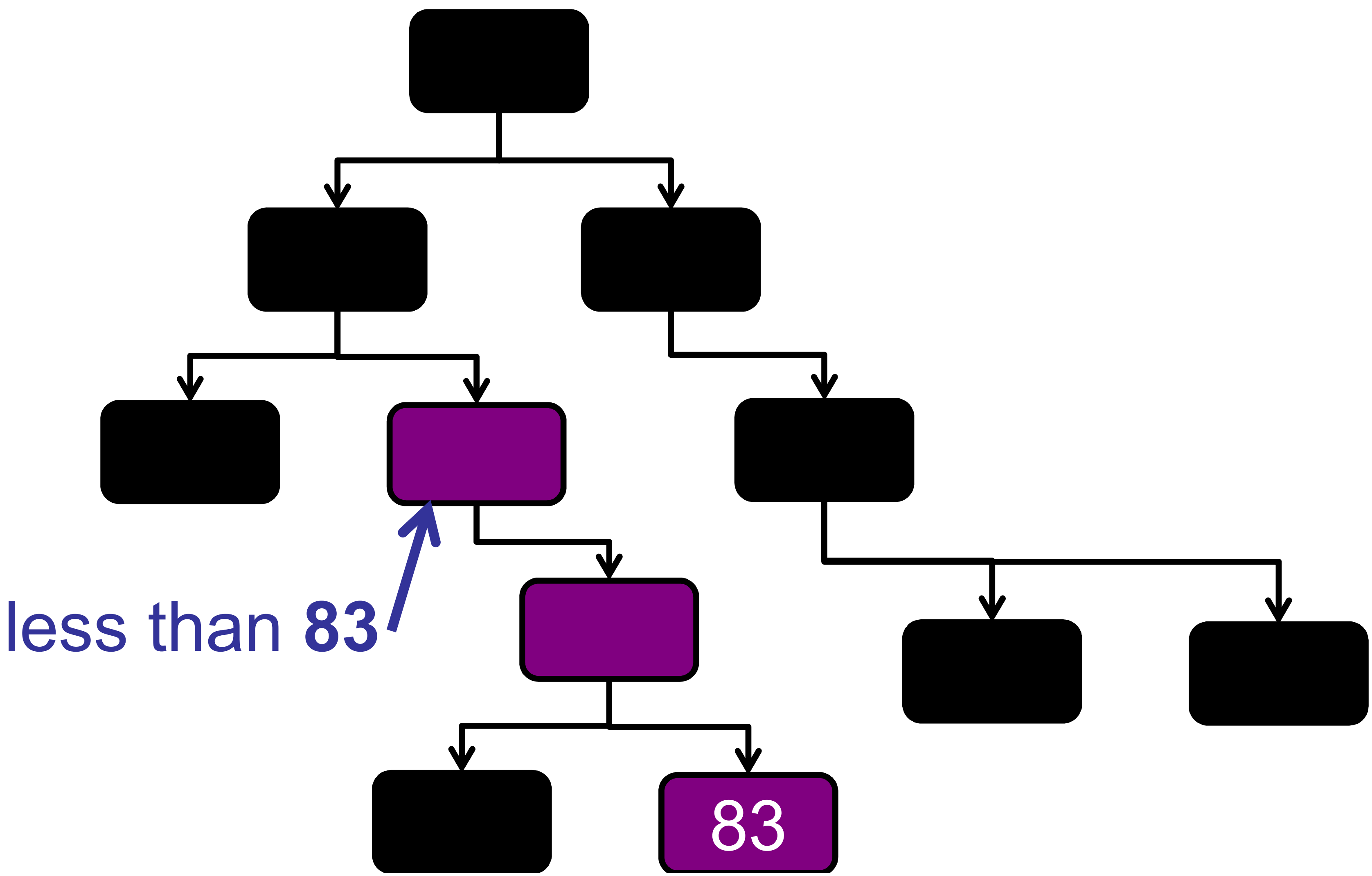
Binary Search Tree

successor()



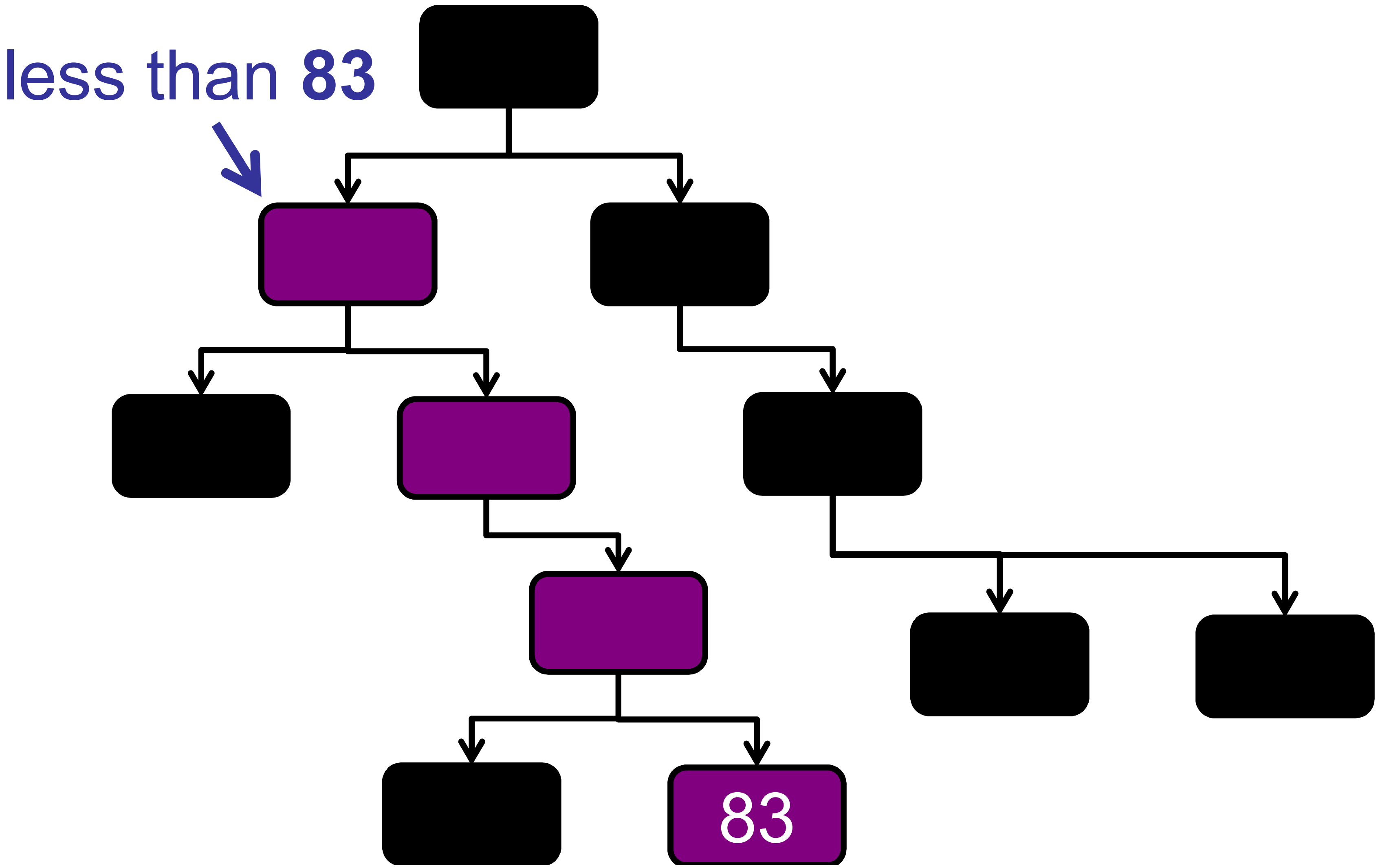
Binary Search Tree

successor()



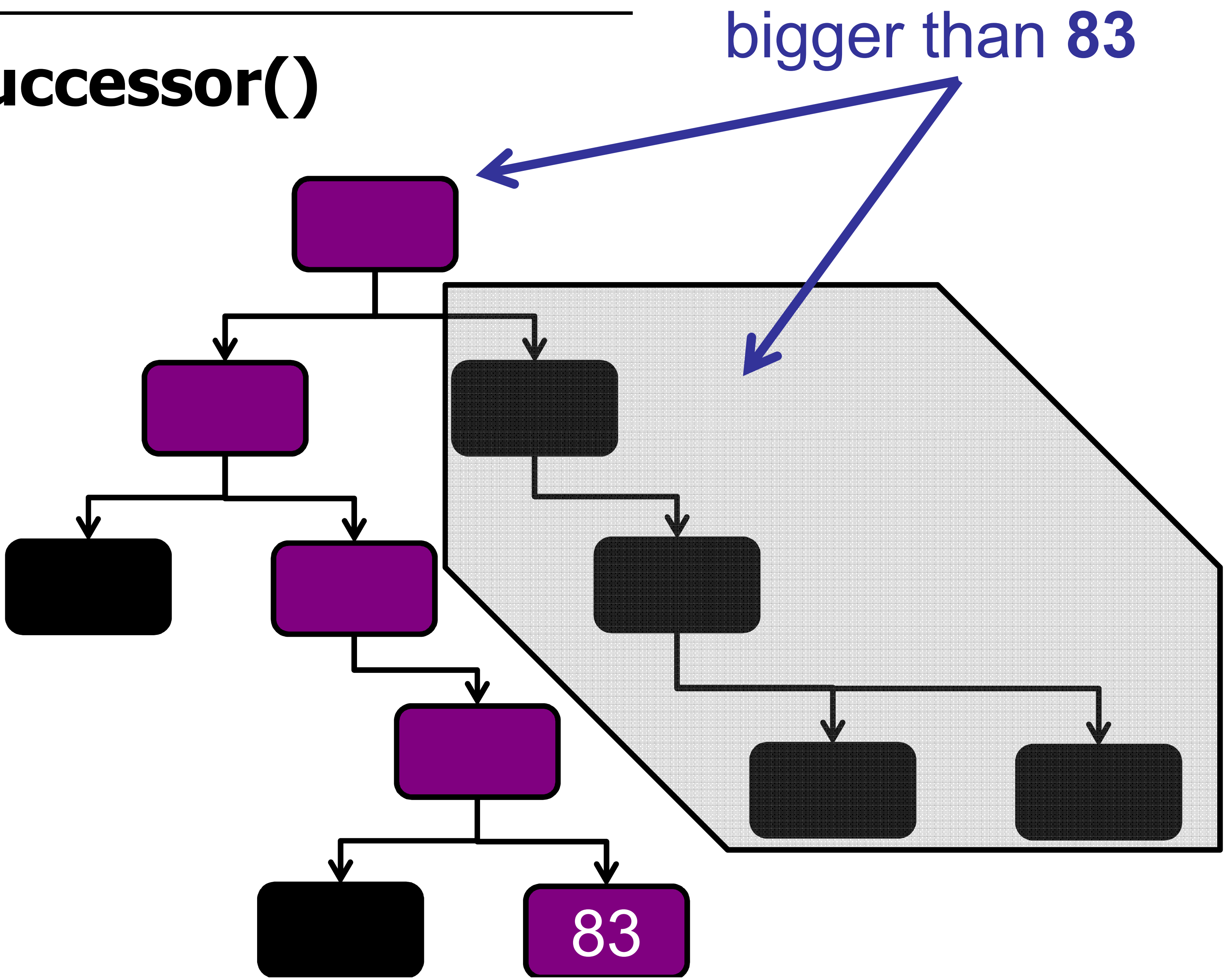
Binary Search Tree

successor()



Binary Search Tree

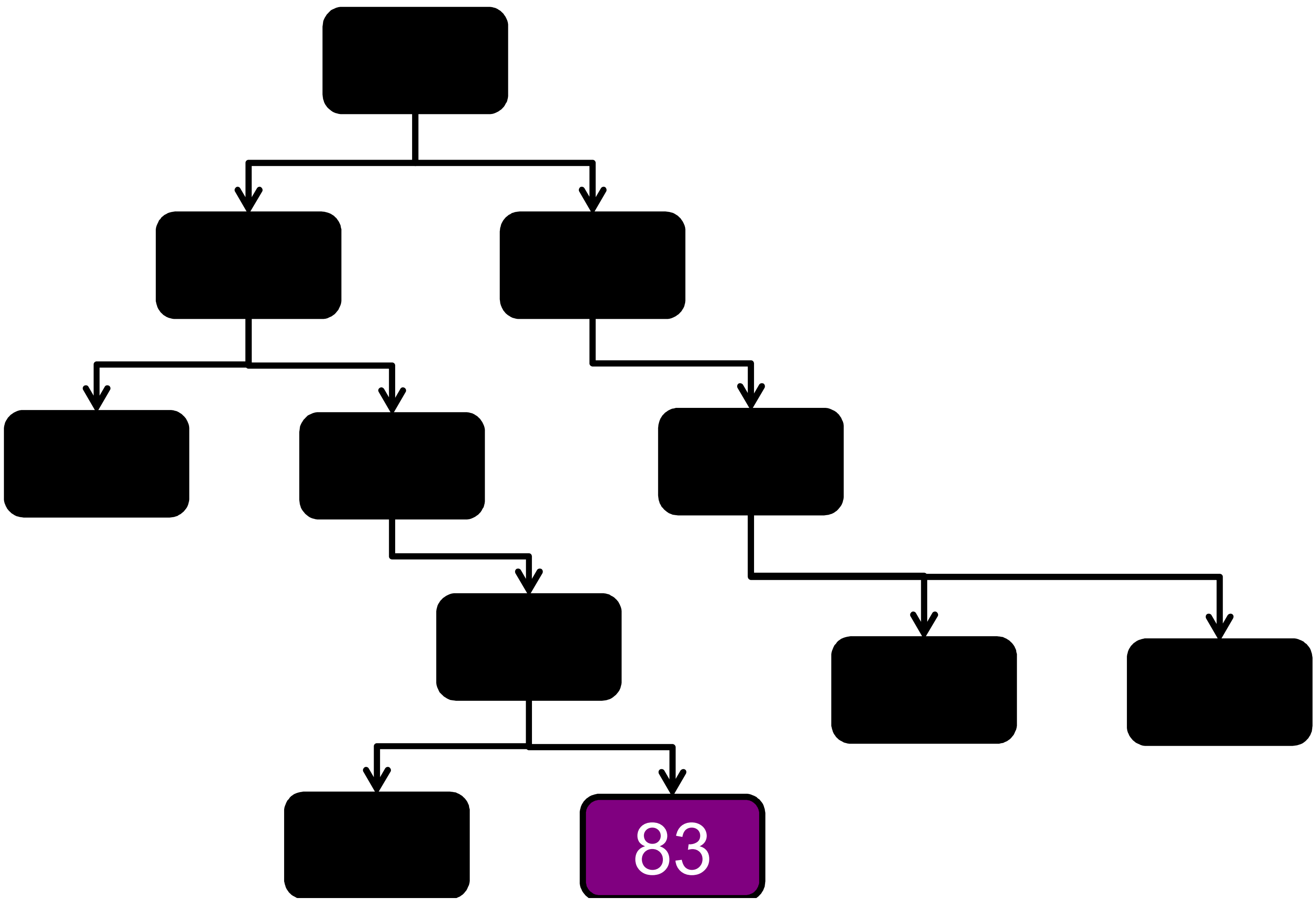
successor()



```
successor(){
  if (right != null) {
      return right.findMin();
  }
  else{
      y = parent;
      x = this;
      while ((y!=null) && (x==y.right)){
              x = y;
              y = x.parent;
      }
      return y;
  }
```

Binary Search Tree

successor()



Airport Scheduling

Dynamic Dictionary

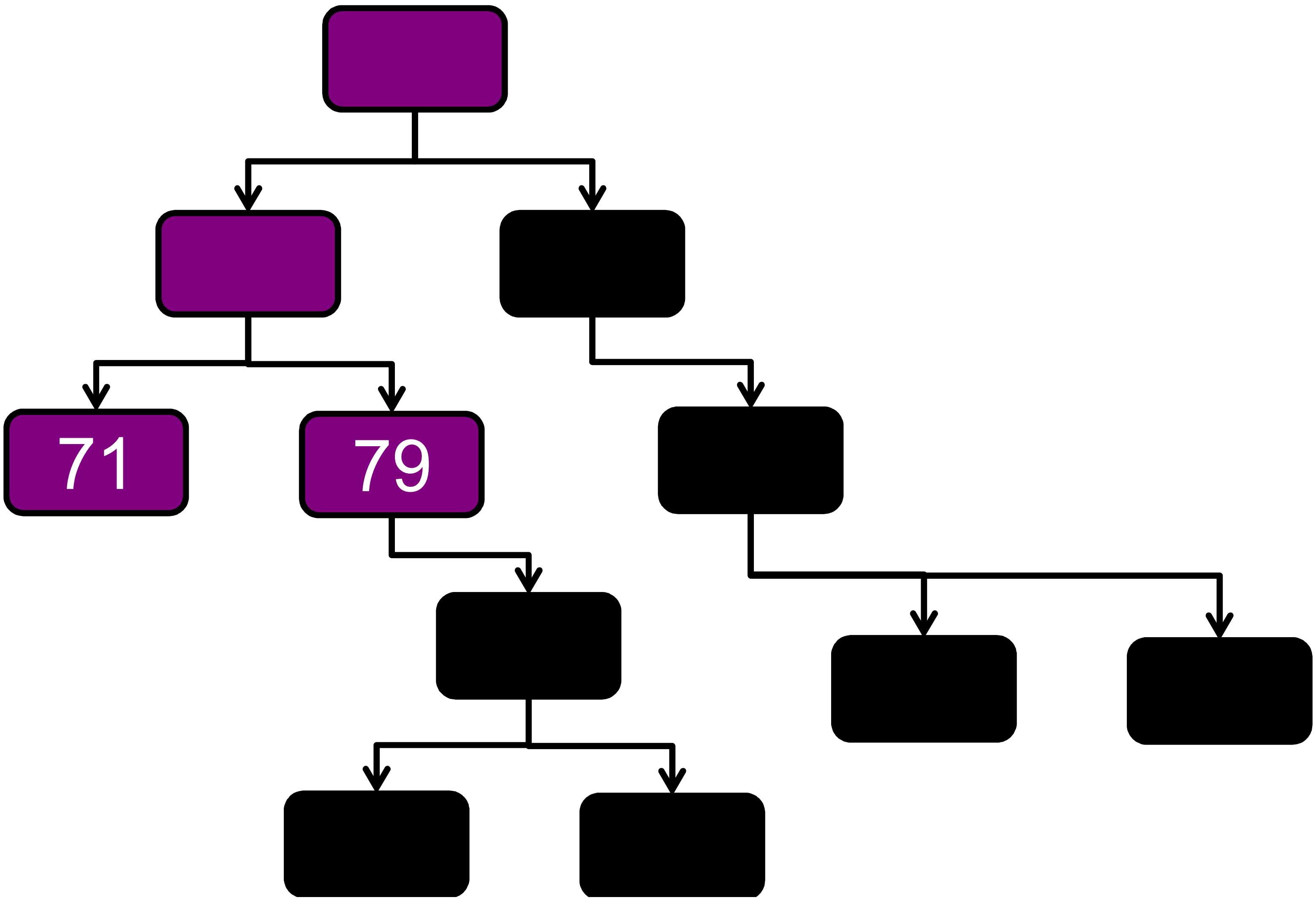
6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

- search-successor(8:24) = 12:21

How do we implement this?

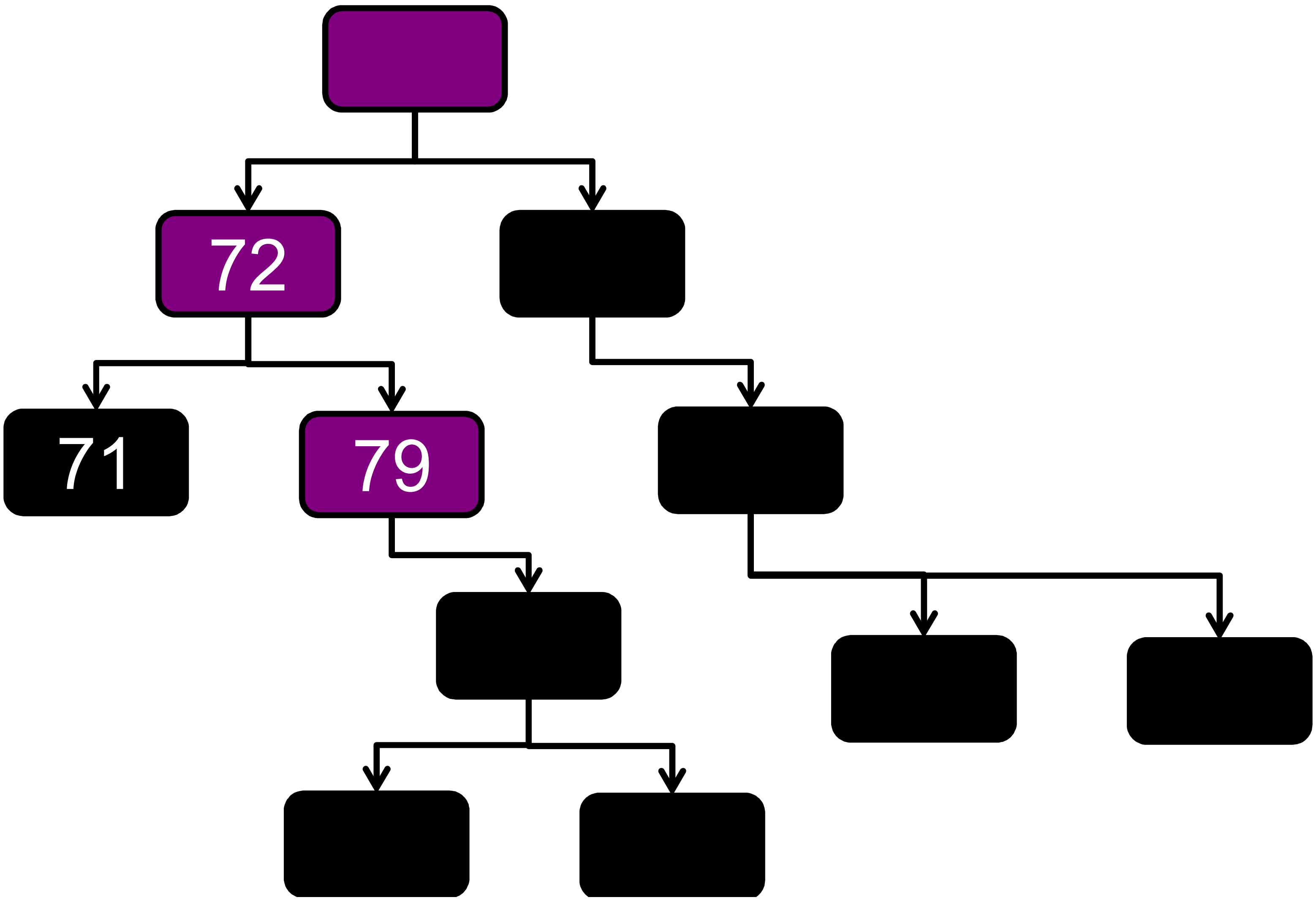
Binary Search Tree

search-successor(73)



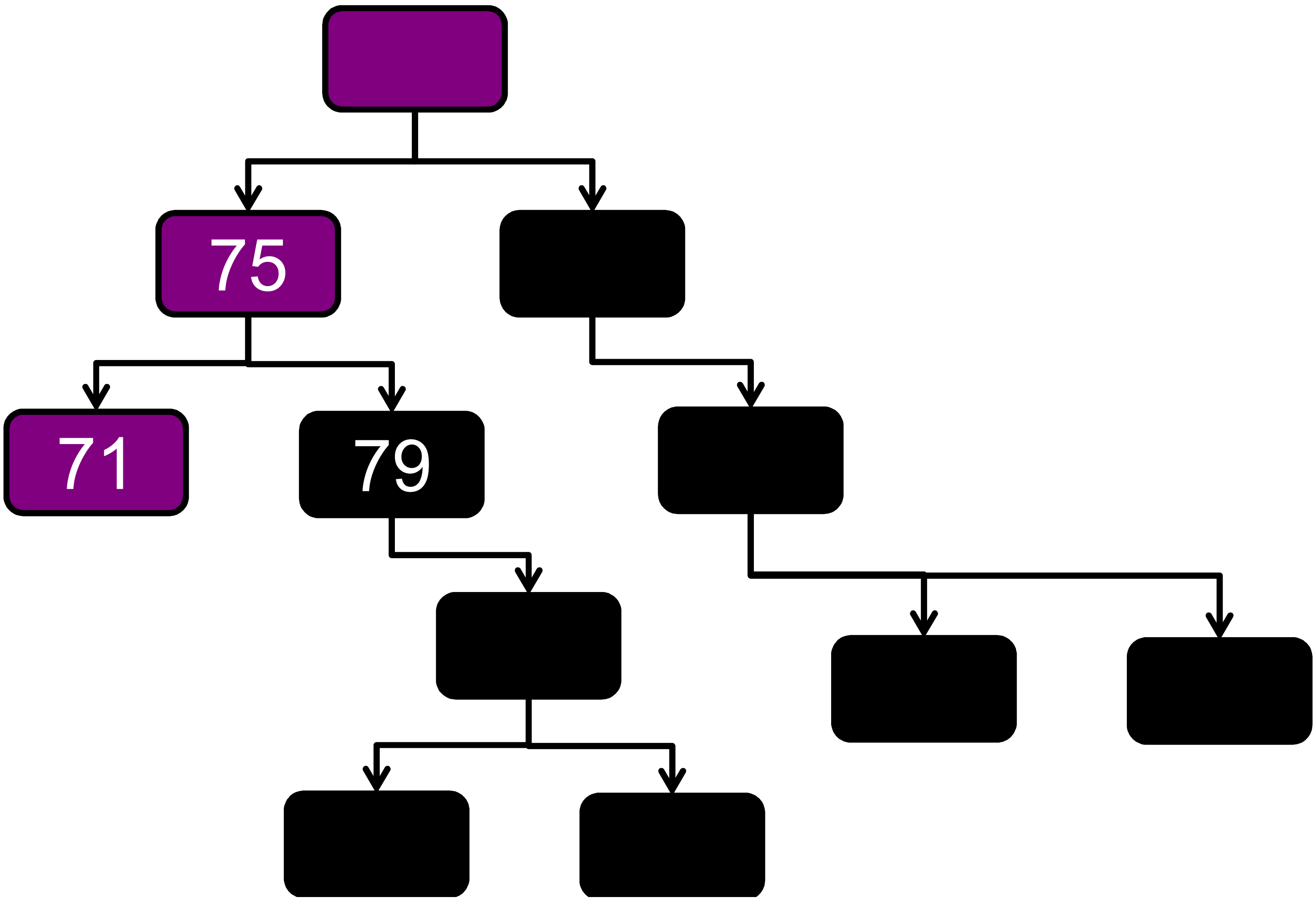
Binary Search Tree

search-successor(73)



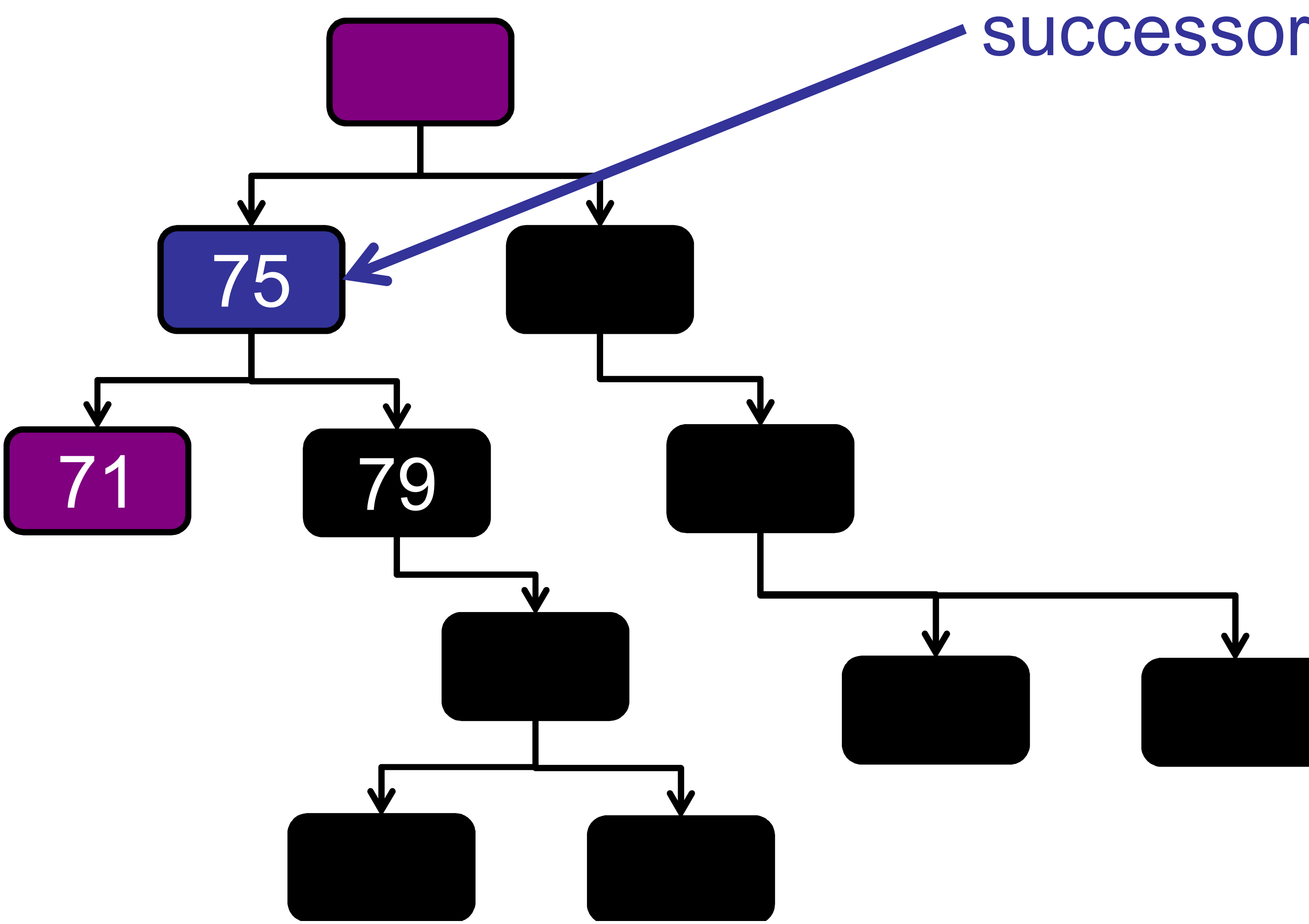
Binary Search Tree

search-successor(73)



Binary Search Tree

search-successor(73)



Binary Search Tree

Time: O(h)

```
search-successor(int key){
   x = search-node(key);
   if (key < x.key){
        return x.key;
   }
   else{
        succ = x.successor();
        return succ.key;
   }
}
```

returns last node discovered during failed search

Binary Search Tree

Summary:

- search: O(h)
- insert: O(h)

Other operations: O(h)

- findMax
- findMin
- successor, search-successor
- predecessor, search-predecessor
- delete