More Data Structures (Part 2)

Queues

Queue

Queue

Queue Operations

- classically, queues only support two operations

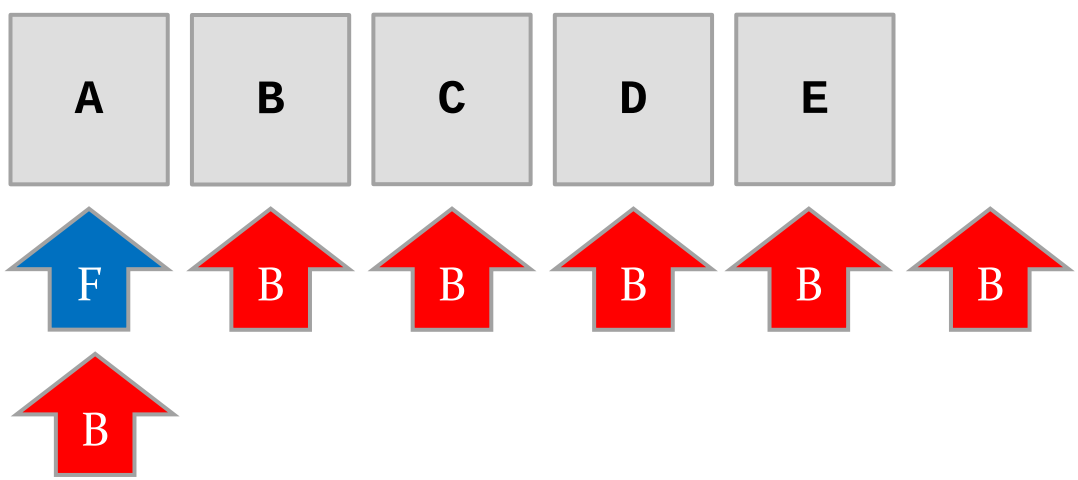
1. enqueue
   - add to the back of the queue
2. dequeue
   - remove from the front of the queue

Queue Optional Operations

- optional operations
  1. size
     - number of elements in the queue
  2. isEmpty
     - is the queue empty?
  3. peek
     - get the front element (without removing it)
  4. search
     - find the position of the element in the queue
  5. isFull
     - is the queue full? (for queues with finite capacity)
  6. capacity
     - total number of elements the queue can hold (for queues with finite capacity)

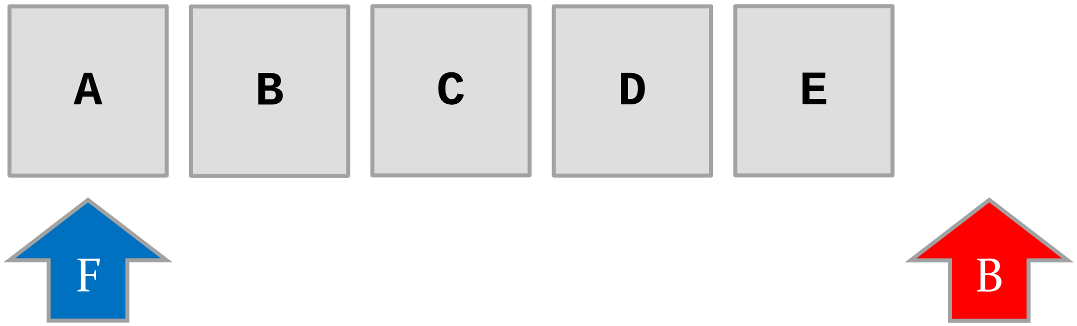
Enqueue

1. `q.enqueue("A")`
2. `q.enqueue("B")`
3. `q.enqueue("C")`
4. `q.enqueue("D")`
5. `q.enqueue("E")`



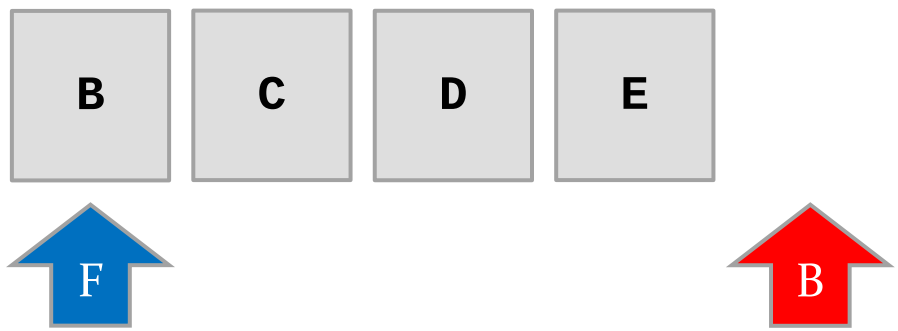
Dequeue

1. `String s = q.dequeue()`



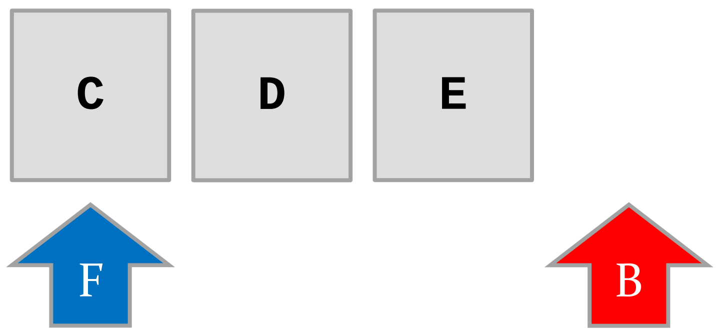
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`



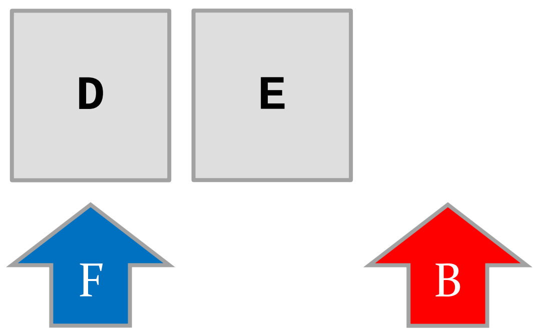
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`



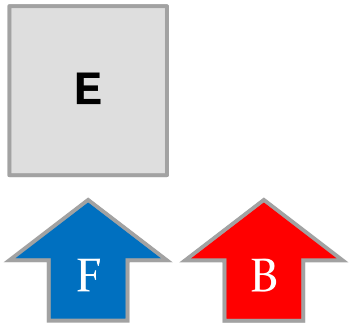
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`
4. `s = q.dequeue()`



Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`
4. `s = q.dequeue()`
5. `s = q.dequeue()`



FIFO

- queue is a First-In-First-Out (FIFO) data structure
  - the first element enqueued in the queue is the first element that can be accessed from the queue

Implementation with LinkedList

- a linked list can be used to efficiently implement a queue as long as the linked list keeps a reference to the last node in the list
  - required for enqueue
- the head of the list becomes the front of the queue
  - removing (dequeue) from the head of a linked list requires O(1) time
  - adding (enqueue) to the end of a linked list requires O(1) time if a reference to the last node is available

- java.util.LinkedList is a doubly linked list that holds a reference to the last node

```
public class Queue<E> {
  private LinkedList<E> q;

  public Queue() {
    this.q = new LinkedList<E>();
  }

  public enqueue(E element) {
    this.q.addLast(element);
  }

  public E dequeue() {
    return this.q.removeFirst();
  }
}
```

Implementation with LinkedList

- note that there is no need to implement your own queue as there is an existing interface
  - the interface does not use the names enqueue and dequeue however

java.util.Queue

```
public interface Queue<E>
  extends Collection<E>
```

`boolean`	`add(E e)` Inserts the specified element into this queue…
`E`	`remove()` Retrieves and removes the head of this queue…
`E`	`peek()` Retrieves, but does not remove, the head of this queue…

- plus other methods
  - http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

java.util.Queue

- **`LinkedList`** implements **`Queue`** so if you ever need a queue you can simply use:
  - e.g. for a queue of strings

```
Queue<String> q = new LinkedList<String>();
```

Queue applications

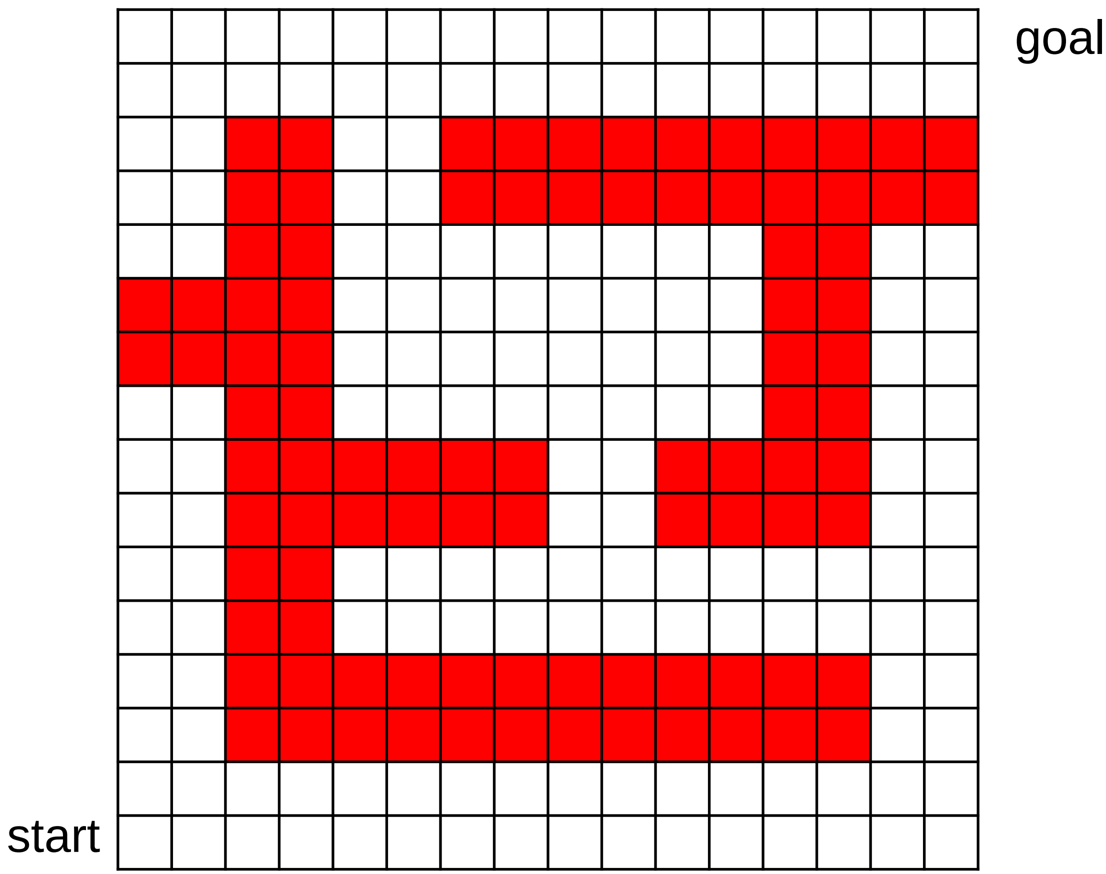
- queues are useful whenever you need to hold elements in their order of arrival
  - serving requests of a single resource
    - printer queue
    - disk queue
    - CPU queue
    - web server

Robotics example

- in robotics, the path planning problem is
  - given a map of the environment, find a path between the starting point of the robot and a goal location that does not pass through any obstacles

- one approach is to use a grid for the map

Grid-based map



Wave-front planner

- the wave-front planner finds a path between a start and goal point in spaces represented as a grid where
  - free space is labeled with a 0
  - obstacles are labeled with a 1
  - the goal is labeled with a 2
  - the start is known

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

start

Wave-front planner

- starting with the goal cell

```
label L = 2
while start cell is unlabelled {
  for each cell C with label L {
    for each cell Z connected to C with label 0 {
      label Z with L+1
    }
  }
  L = L + 1
}
```

Wave-front planner

	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
start	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3	2
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
start	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3	2
	0	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
goal	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	0	0	0	14	13	12	11	10	9	8	7	6	5	4	3	2
	0	0	0	0	14	13	12	11	10	9	8	7	6	5	4	3
	0	0	1	1	0	14	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
start	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
	19	18	1	1	15	14	1	1	1	1	1	1	1	1	1	1
	20	19	1	1	16	15	1	1	1	1	1	1	1	1	1	1
	0	20	1	1	17	16	17	18	19	20	0	0	1	1	0	0
	1	1	1	1	18	17	18	19	20	0	0	0	1	1	0	0
	1	1	1	1	19	18	19	20	0	0	0	0	1	1	0	0
	0	0	1	1	20	19	20	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
start	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
	19	18	1	1	15	14	1	1	1	1	1	1	1	1	1	1
	20	19	1	1	16	15	1	1	1	1	1	1	1	1	1	1
	21	20	1	1	17	16	17	18	19	20	21	22	1	1	37	38
	1	1	1	1	18	17	18	19	20	21	22	23	1	1	36	37
	1	1	1	1	19	18	19	20	21	22	23	24	1	1	35	36
	0	0	1	1	20	19	20	21	22	23	24	25	1	1	34	35
	0	0	1	1	1	1	1	1	23	24	1	1	1	1	33	34
	0	0	1	1	1	1	1	1	24	25	1	1	1	1	32	33
	0	0	1	1	29	28	27	26	25	26	27	28	29	30	31	32
	0	0	1	1	30	29	28	27	26	27	28	29	30	31	32	33
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	33	34
	0	49	1	1	1	1	1	1	1	1	1	1	1	1	34	35
	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	36
start	*	49	48	47	46	45	44	43	42	41	40	39	38	37	36	37

Wave-front planner

- to generate a path starting from the start point

```
L = start point label
while not at the goal {
  move to any connected cell with label L-1
  L = L-1
}
```

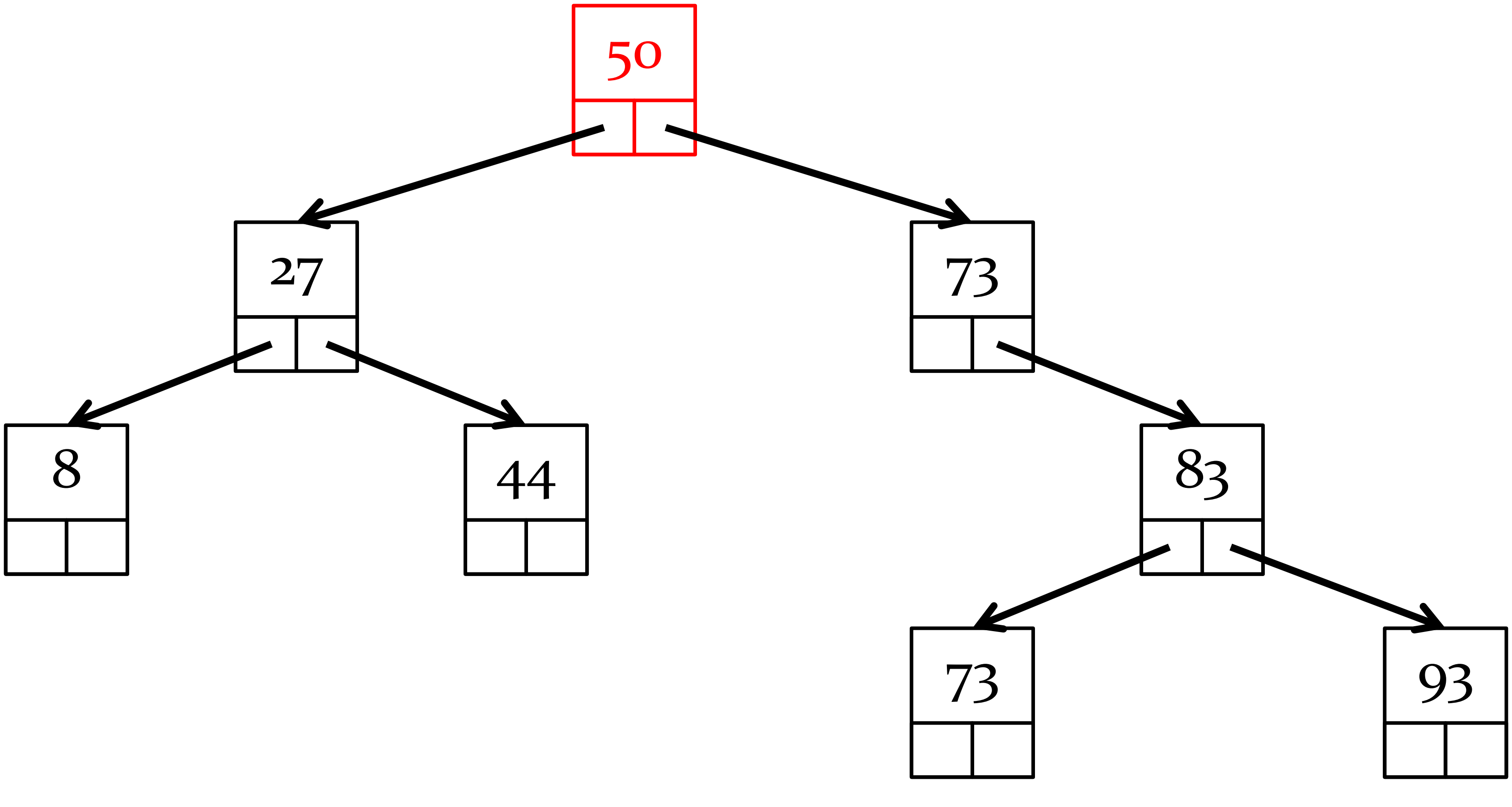
Wave-front planner

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
	19	18	1	1	15	14	1	1	1	1	1	1	1	1	1	1
	20	19	1	1	16	15	1	1	1	1	1	1	1	1	1	1
	21	20	1	1	17	16	17	18	19	20	21	22	1	1	37	38
	1	1	1	1	18	17	18	19	20	21	22	23	1	1	36	37
	1	1	1	1	19	18	19	20	21	22	23	24	1	1	35	36
	0	0	1	1	20	19	20	21	22	23	24	25	1	1	34	35
	0	0	1	1	1	1	1	1	23	24	1	1	1	1	33	34
	0	0	1	1	1	1	1	1	24	25	1	1	1	1	32	33
	0	0	1	1	29	28	27	26	25	26	27	28	29	30	31	32
	0	0	1	1	30	29	28	27	26	27	28	29	30	31	32	33
	0	50	1	1	1	1	1	1	1	1	1	1	1	1	33	34
	50	49	1	1	1	1	1	1	1	1	1	1	1	1	34	35
	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	36
start	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	37

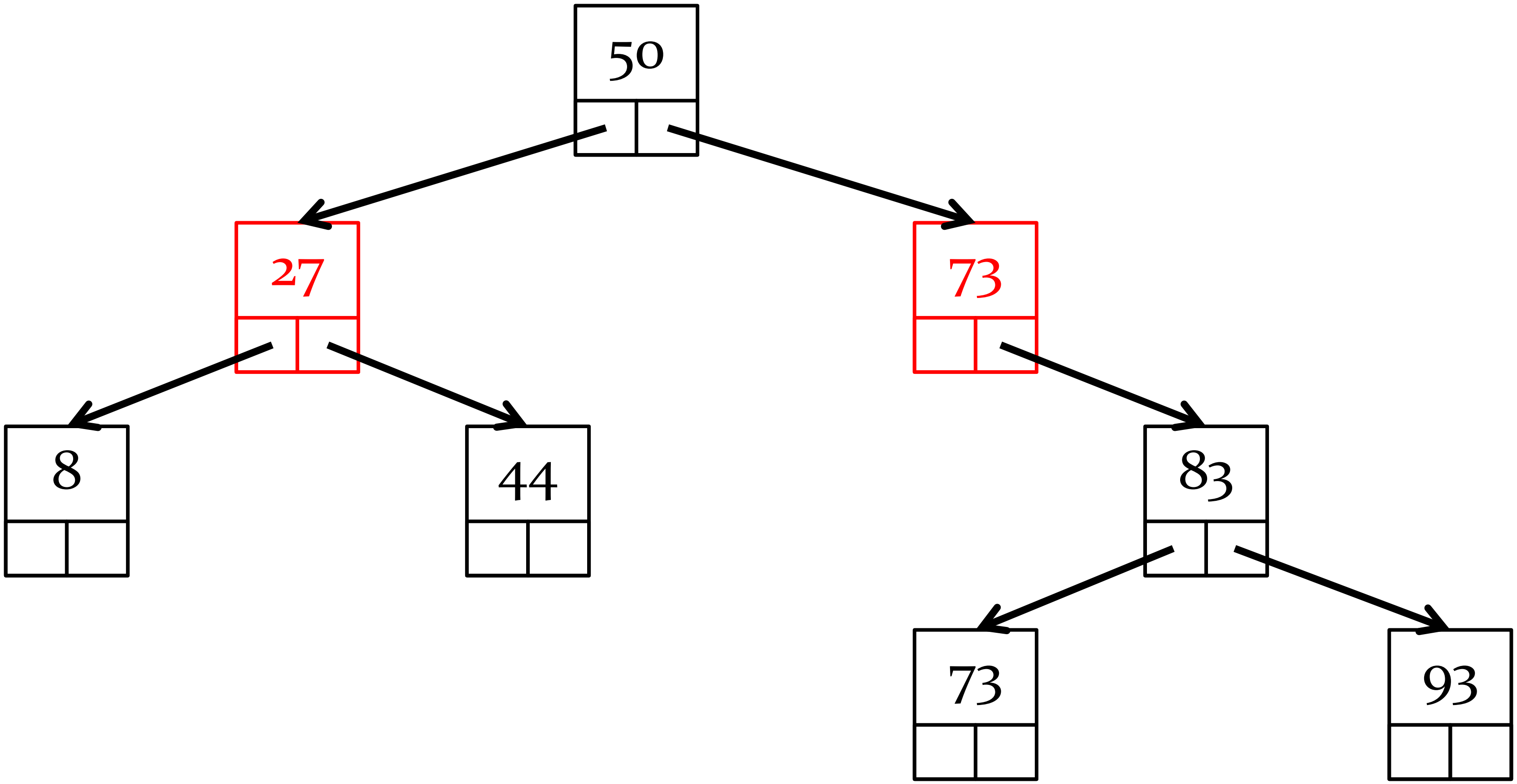
Breadth-first search

- the wave-front planner is actually a classic computer science algorithm called breadth-first search

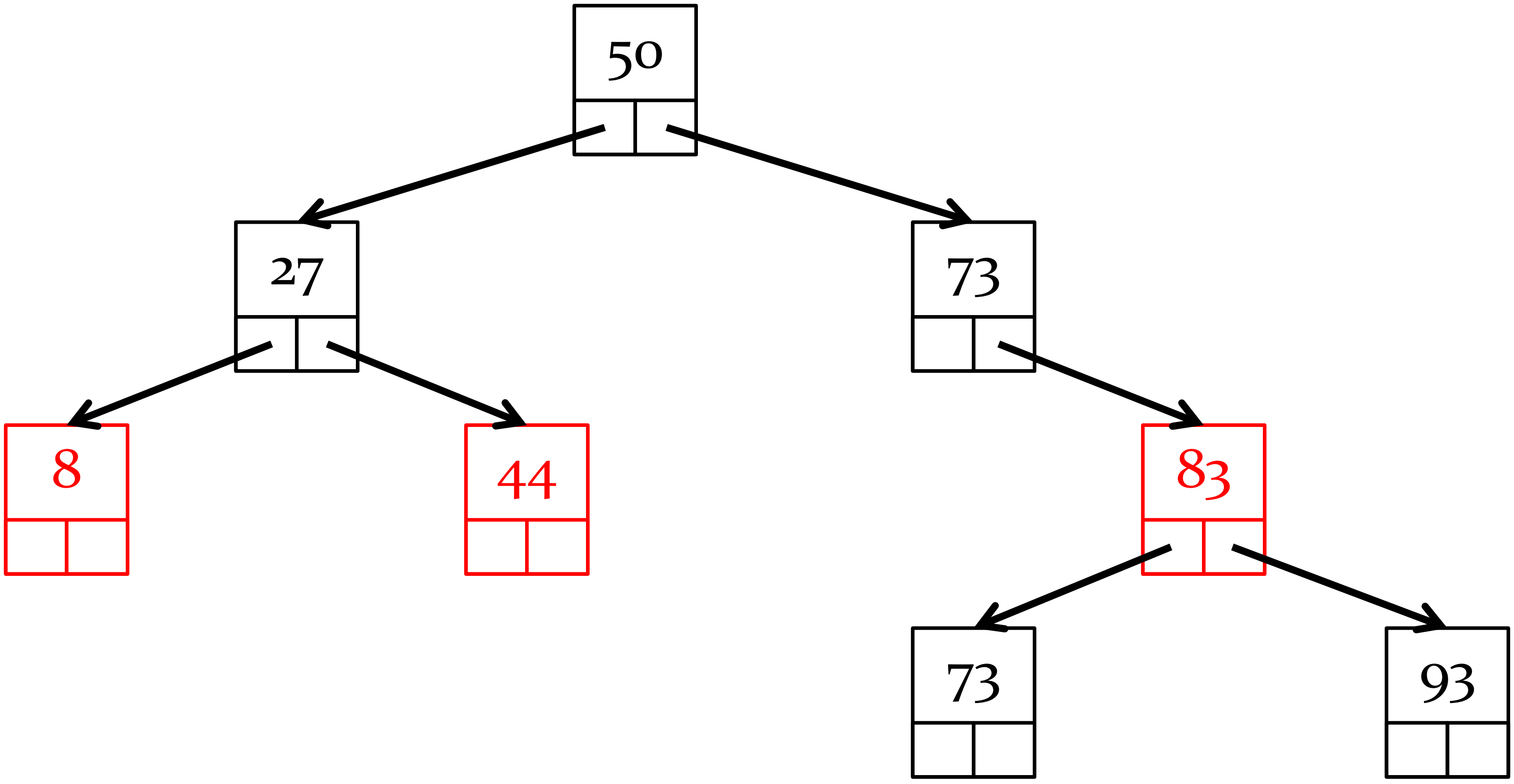
- visiting every node of a tree using breadth-first search results in visiting nodes in order of their level in the tree



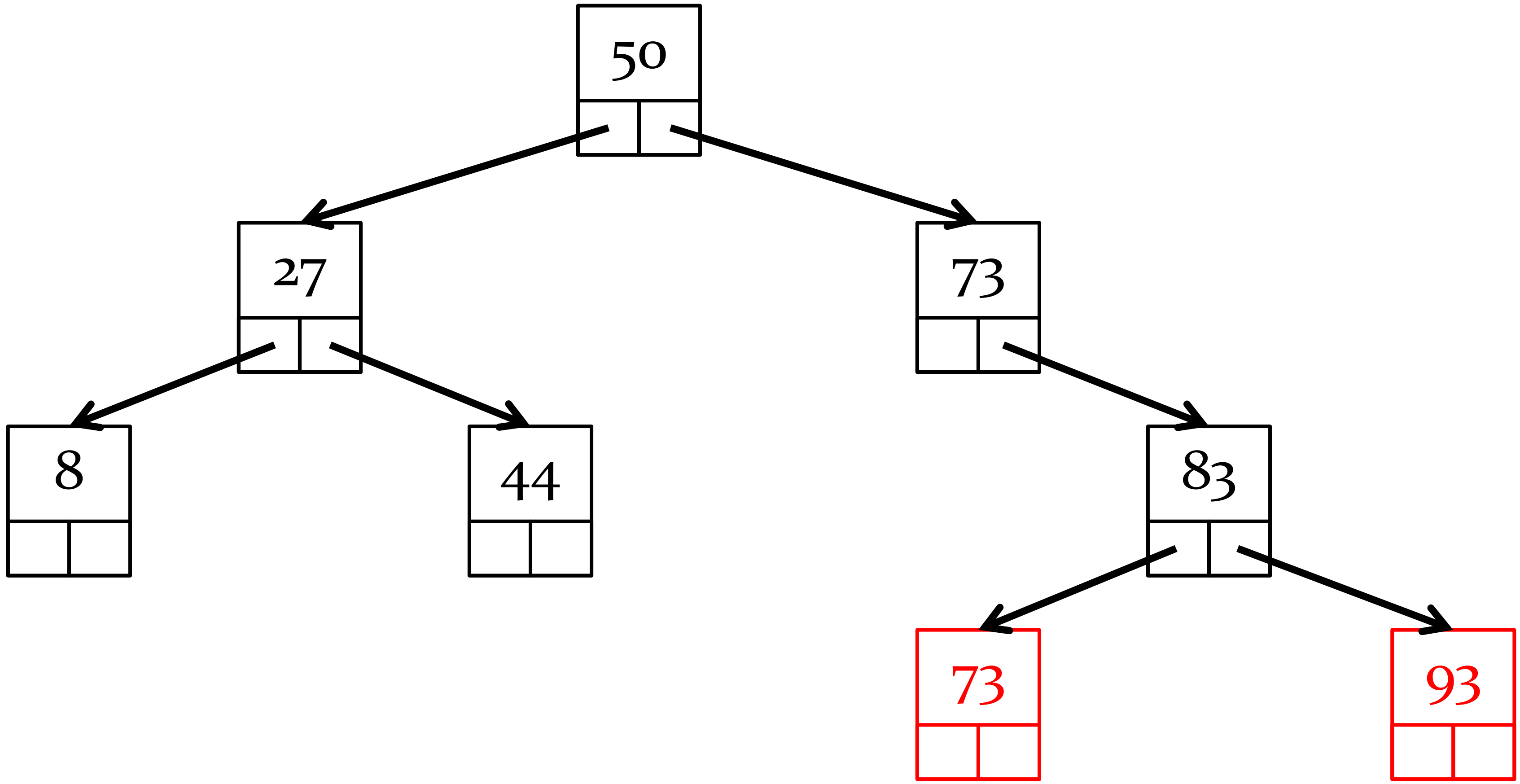
BFS: 50



BFS: 50, 27, 73



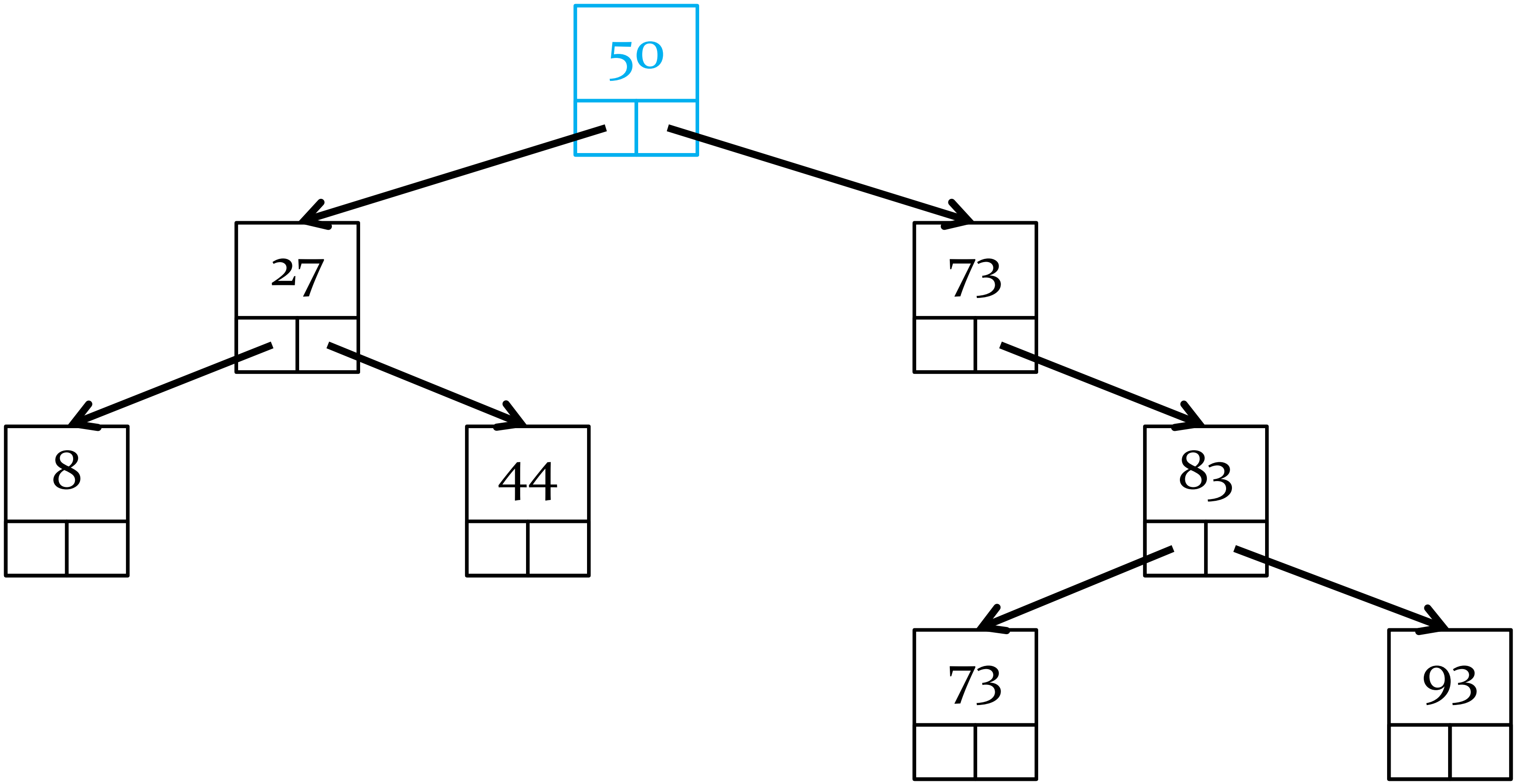
BFS: 50, 27, 73, 8, 44, 83



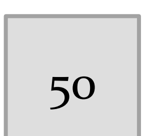
BFS: 50, 27, 73, 8, 44, 83, 73, 93

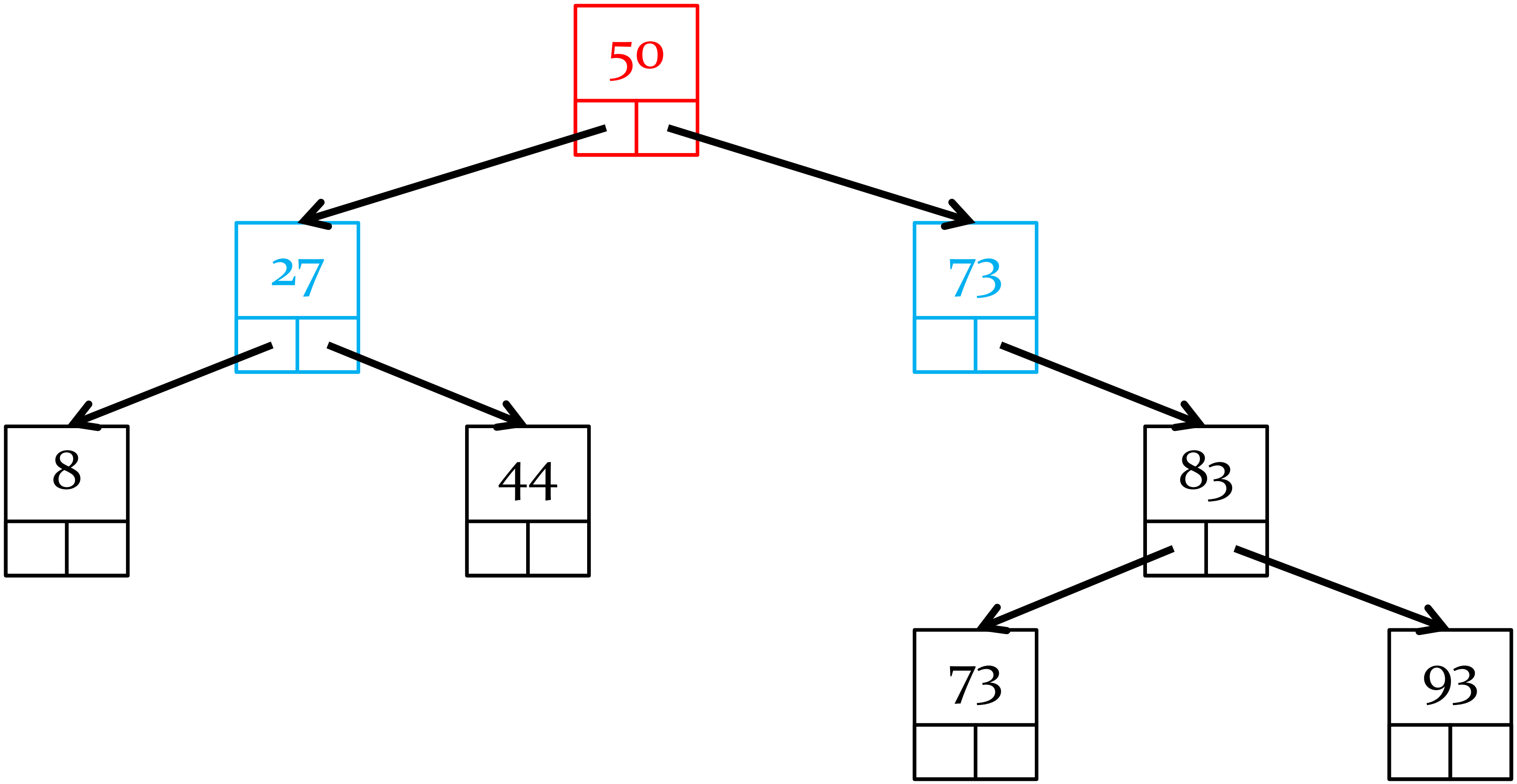
Breadth-first search algorithm

```
Q.enqueue(root node)
while Q is not empty {
  n = Q.dequeue()
  if n.left != null {
    Q.enqueue(n.left)
  }
  if n.right != null {
    Q.enqueue(n.right)
  }
}
```



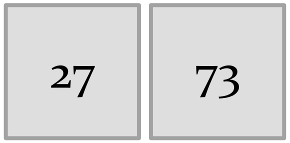
BFS:

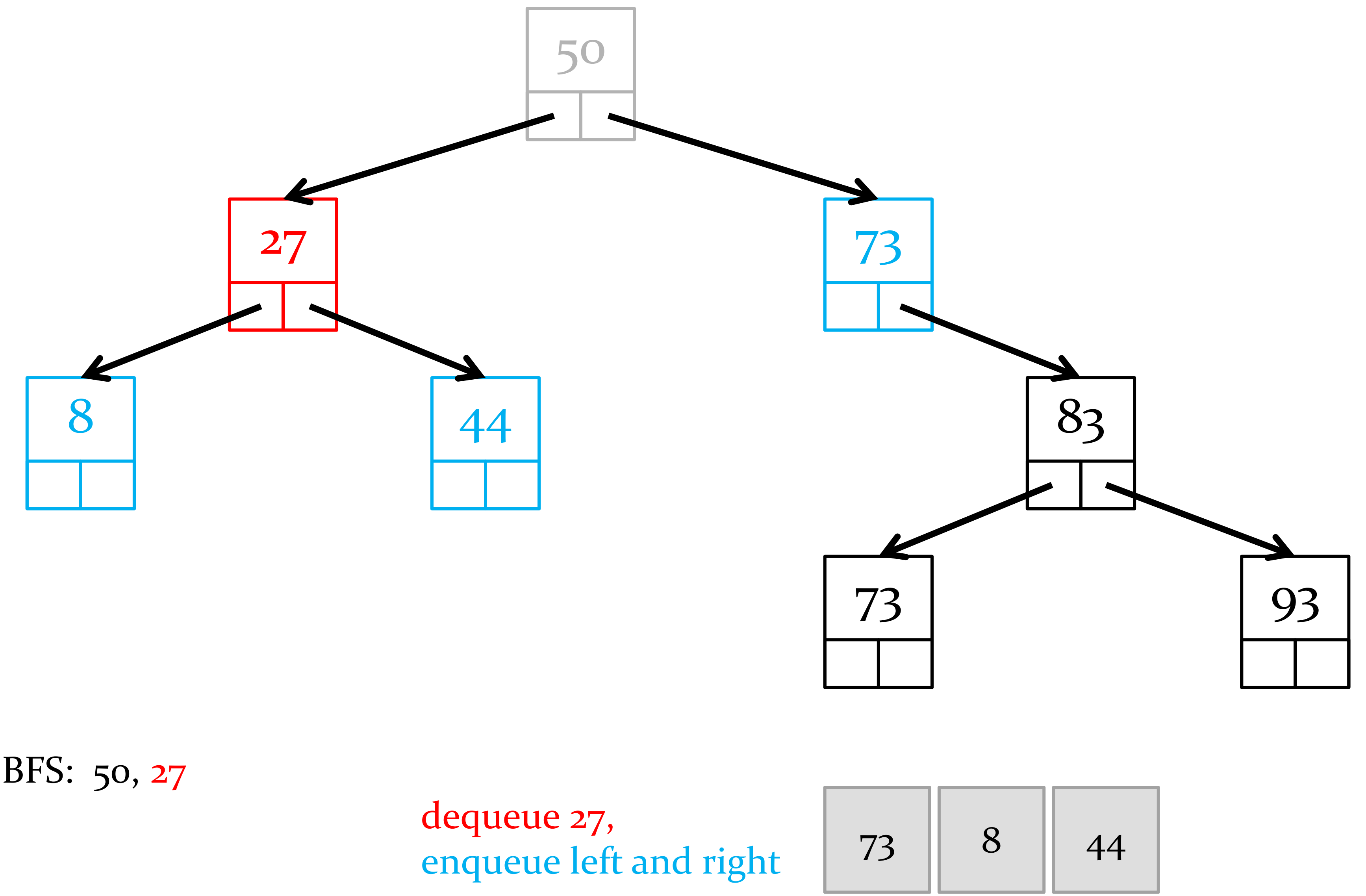


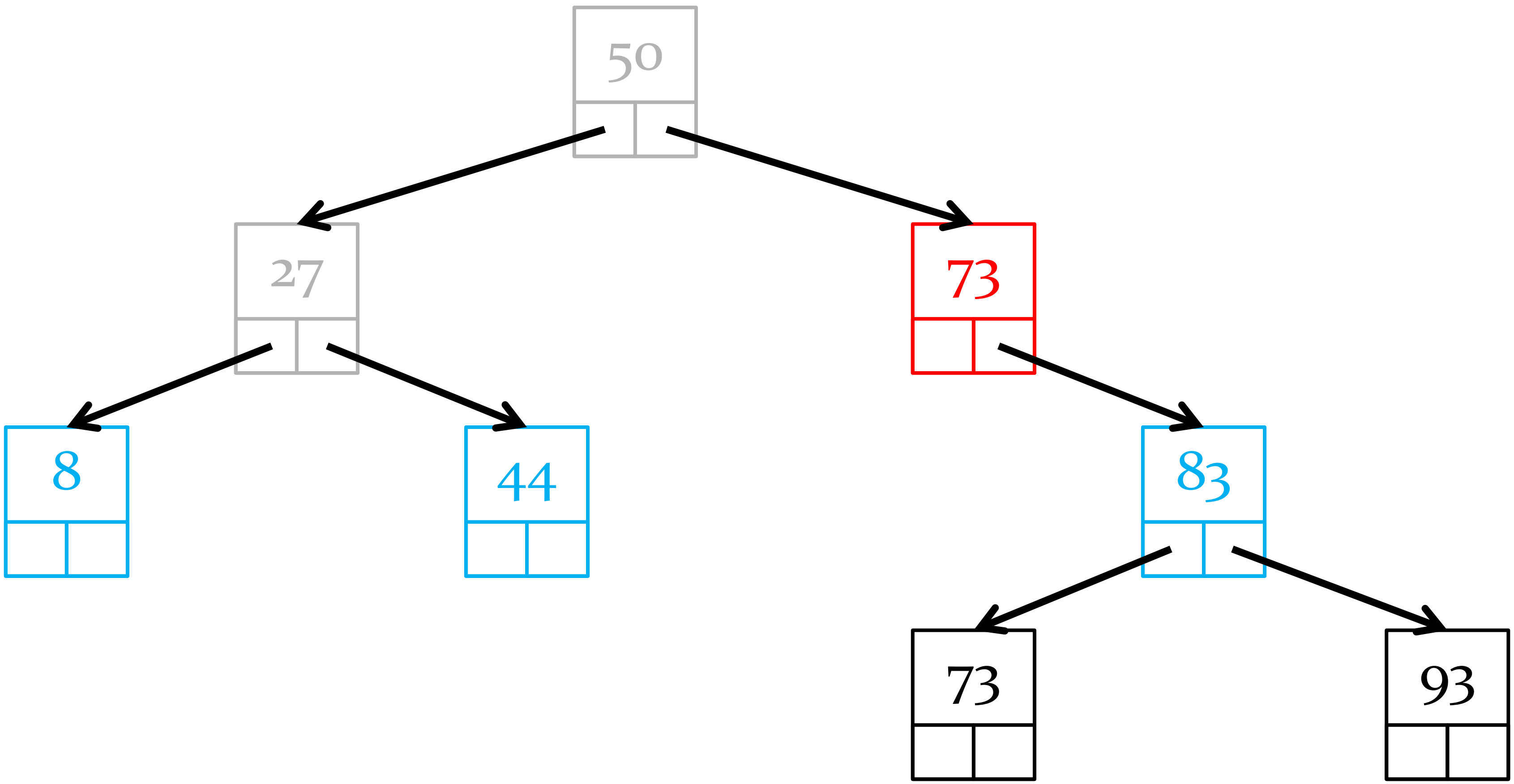


BFS: 50

dequeue 50,
enqueue left and right

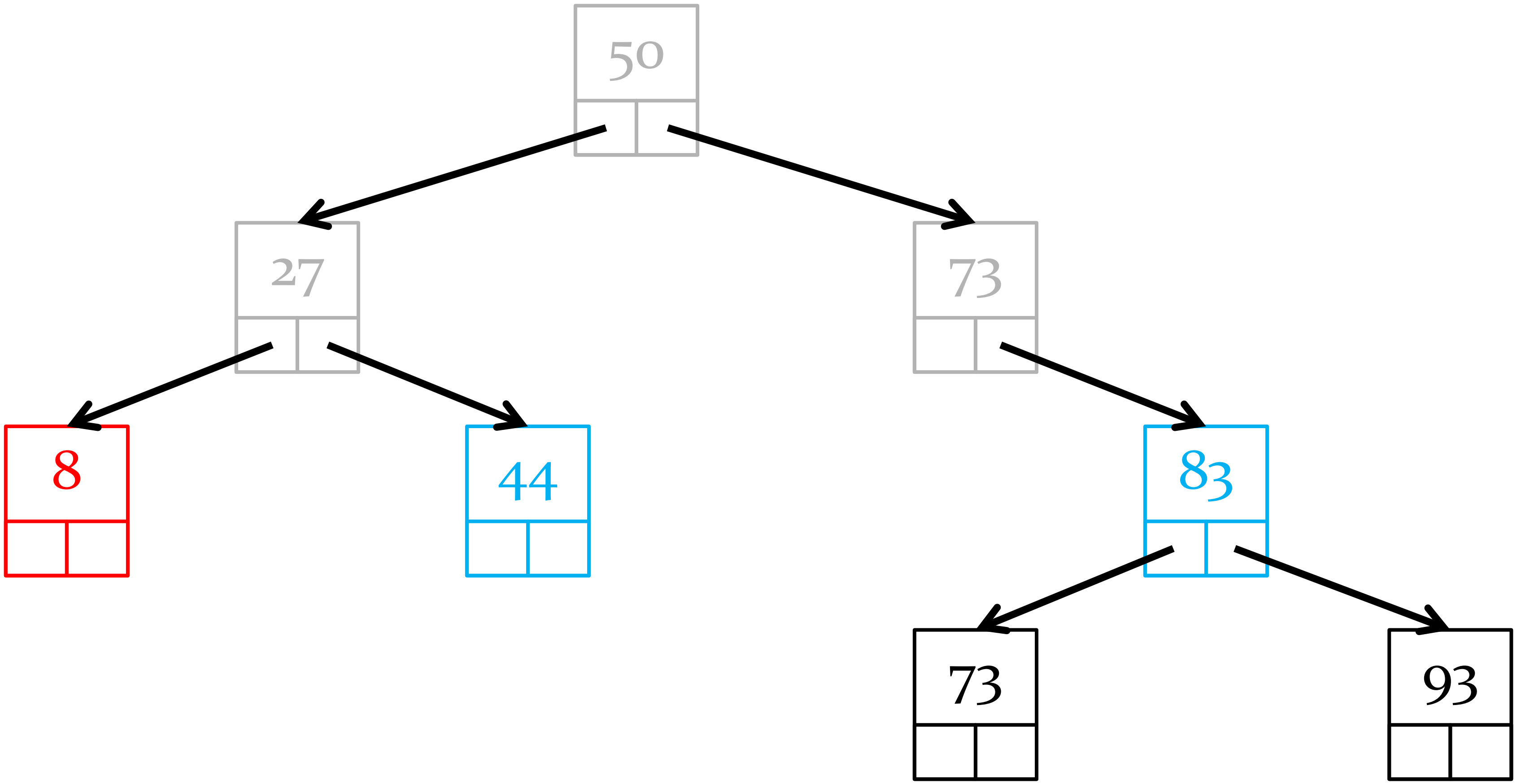



50
27
73
8
44
83
73
93
BFS: 50, 27
dequeue 27,
enqueue left and right
73
8
44



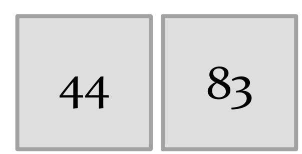
BFS: 50, 27, 73

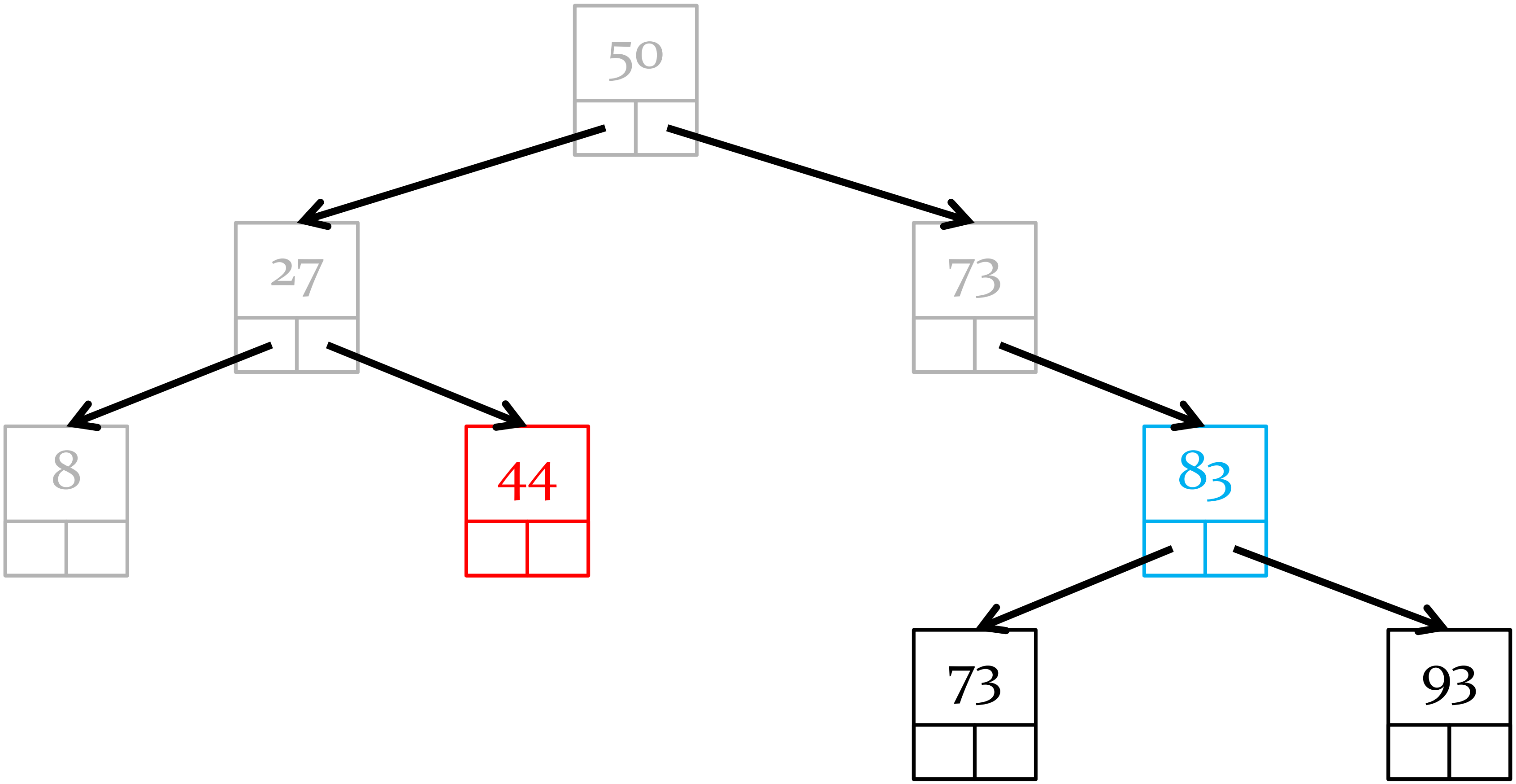
dequeue 73,
enqueue right



BFS: 50, 27, 73, 8

dequeue 8

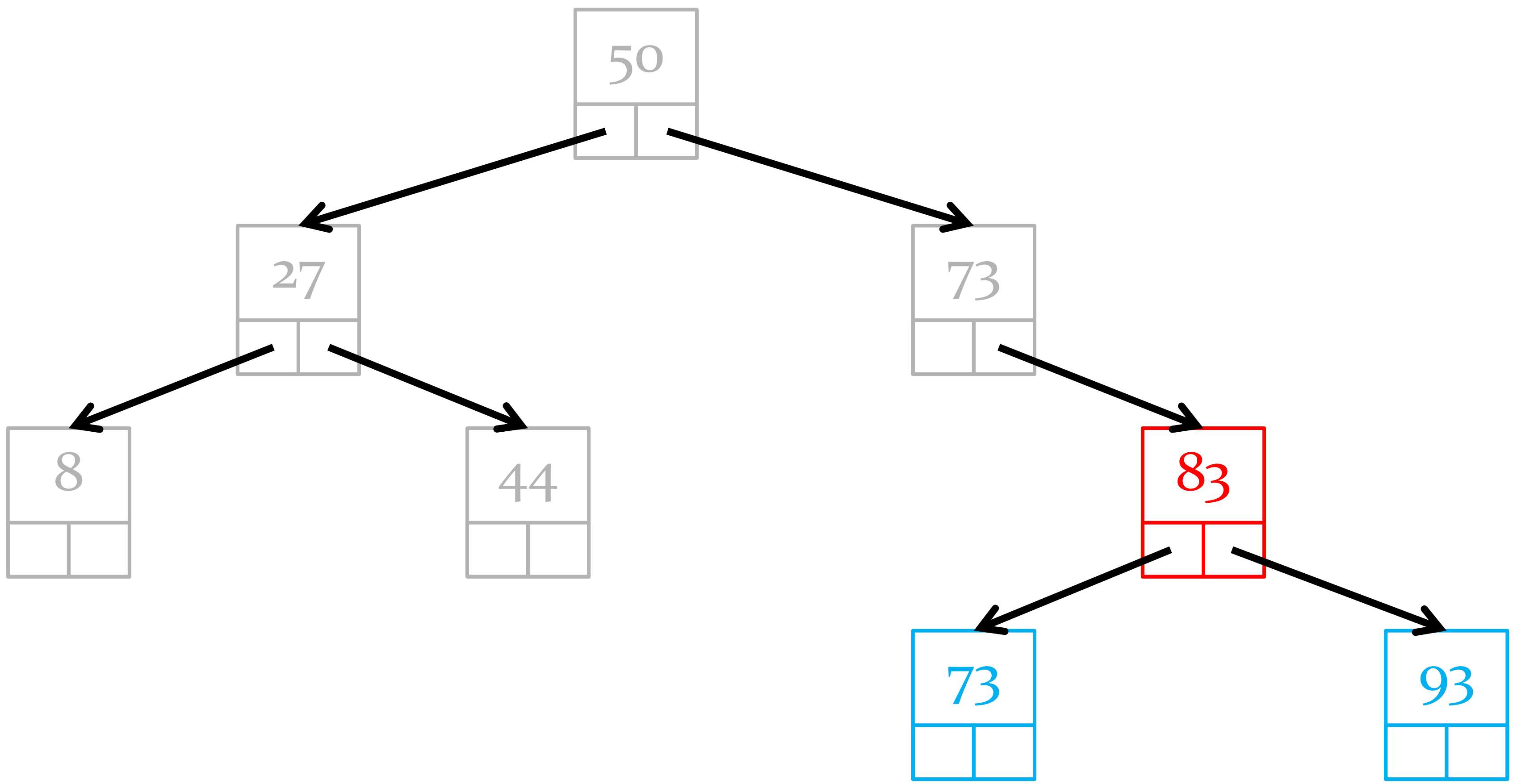




BFS: 50, 27, 73, 8, 44

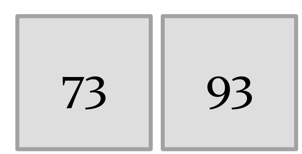
dequeue 44

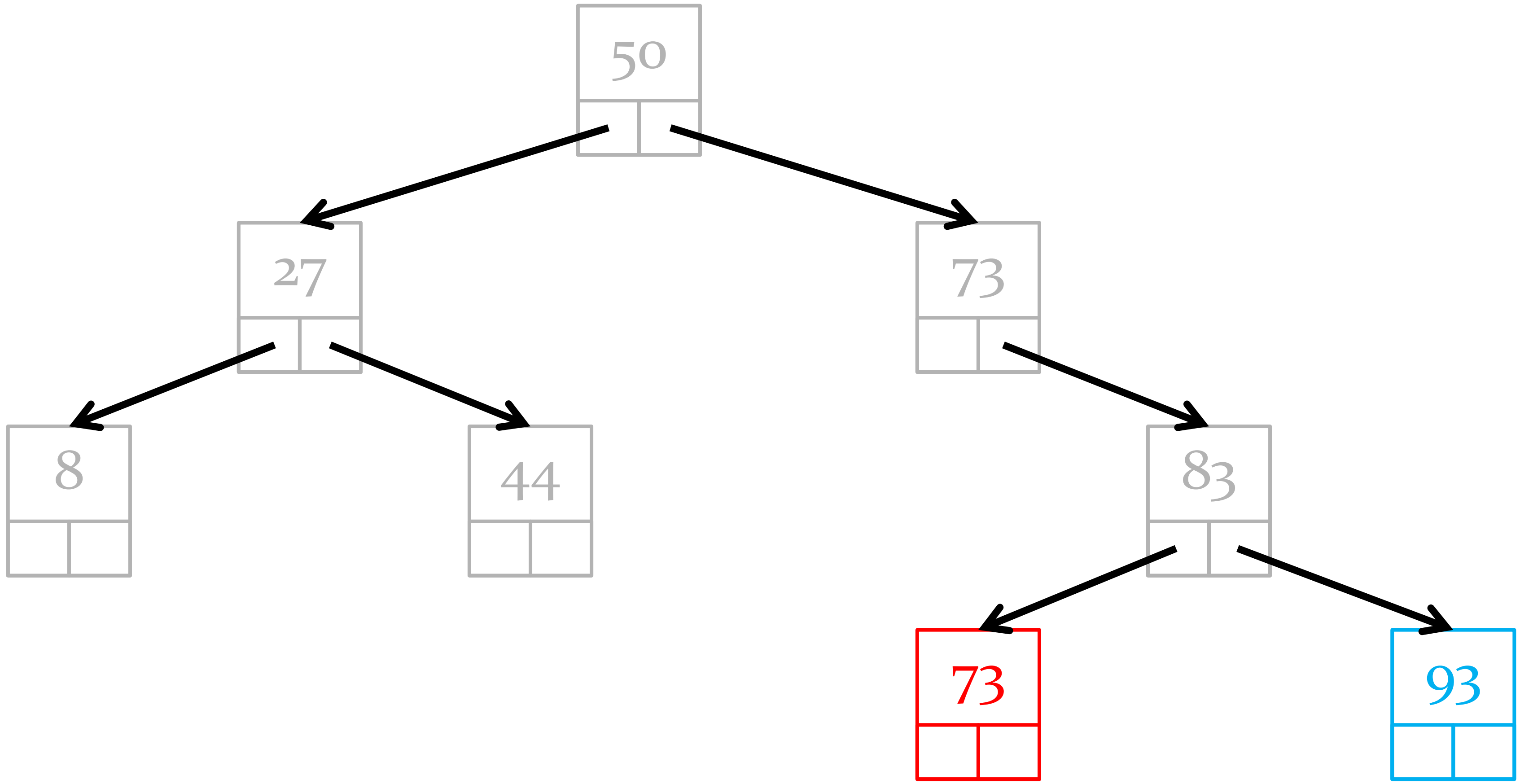
83



BFS: 50, 27, 73, 8, 44, 83

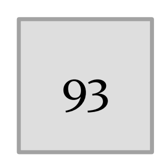
dequeue 83,
enqueue left and right

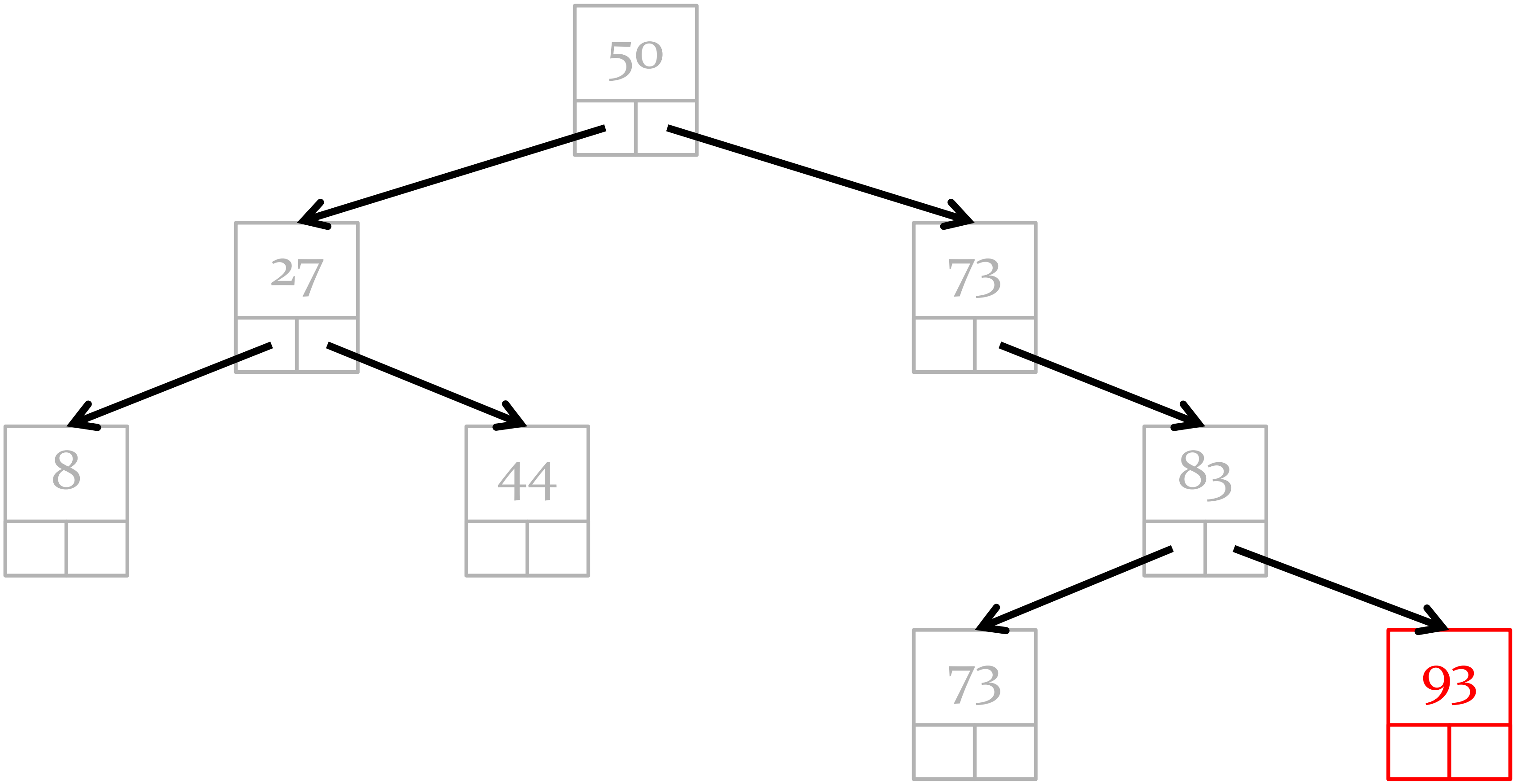




BFS: 50, 27, 73, 8, 44, 83, 73

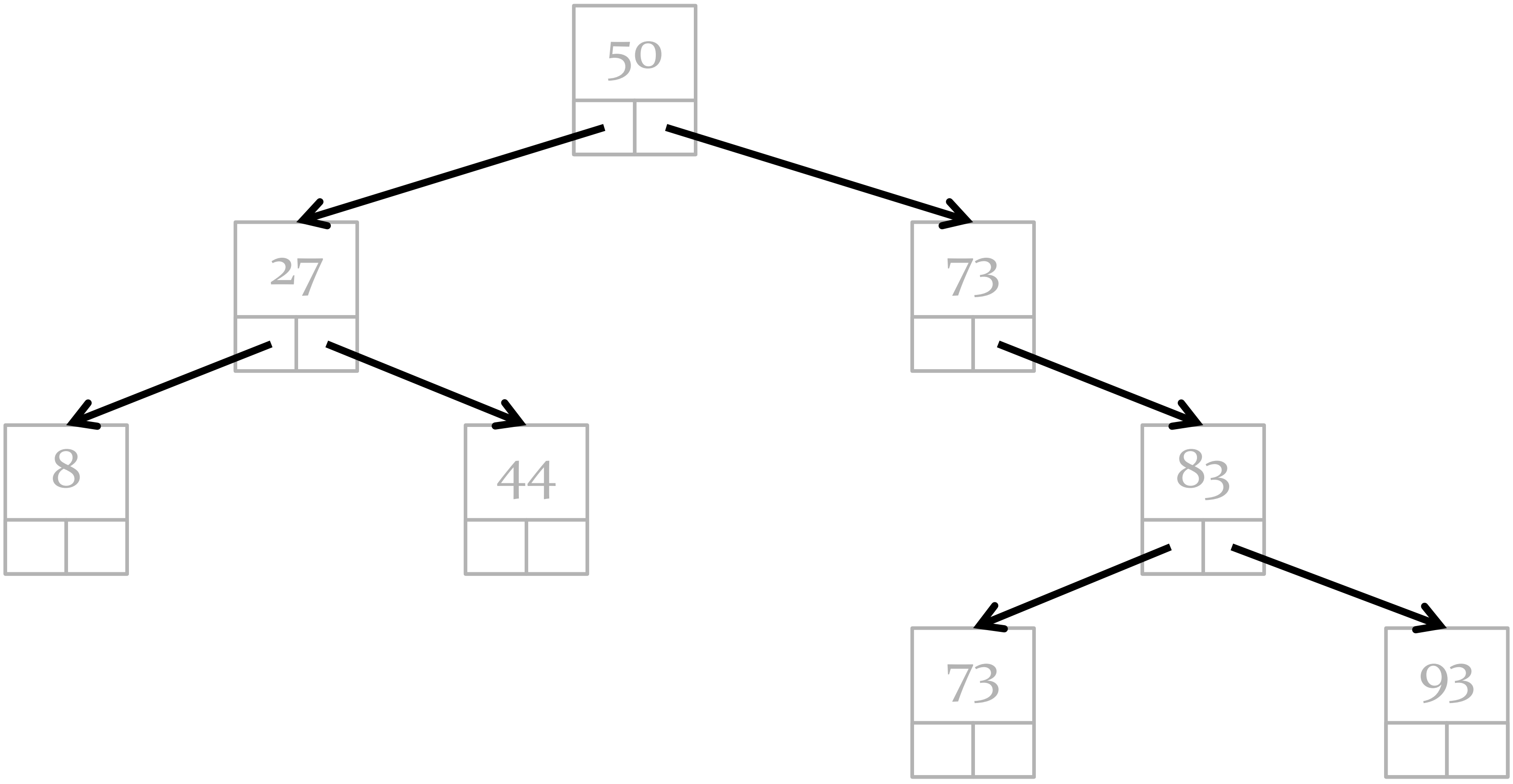
dequeue 73





BFS: 50, 27, 73, 8, 44, 83, 73, 93

dequeue 93



BFS: 50, 27, 73, 8, 44, 83, 73, 93

queue empty

Summary

Major Topics

1. static features (utility classes)
2. non-static features
3. mixing static and non-static features
4. aggregation and composition
5. inheritance
6. graphical user interfaces
7. recursion
8. data structures

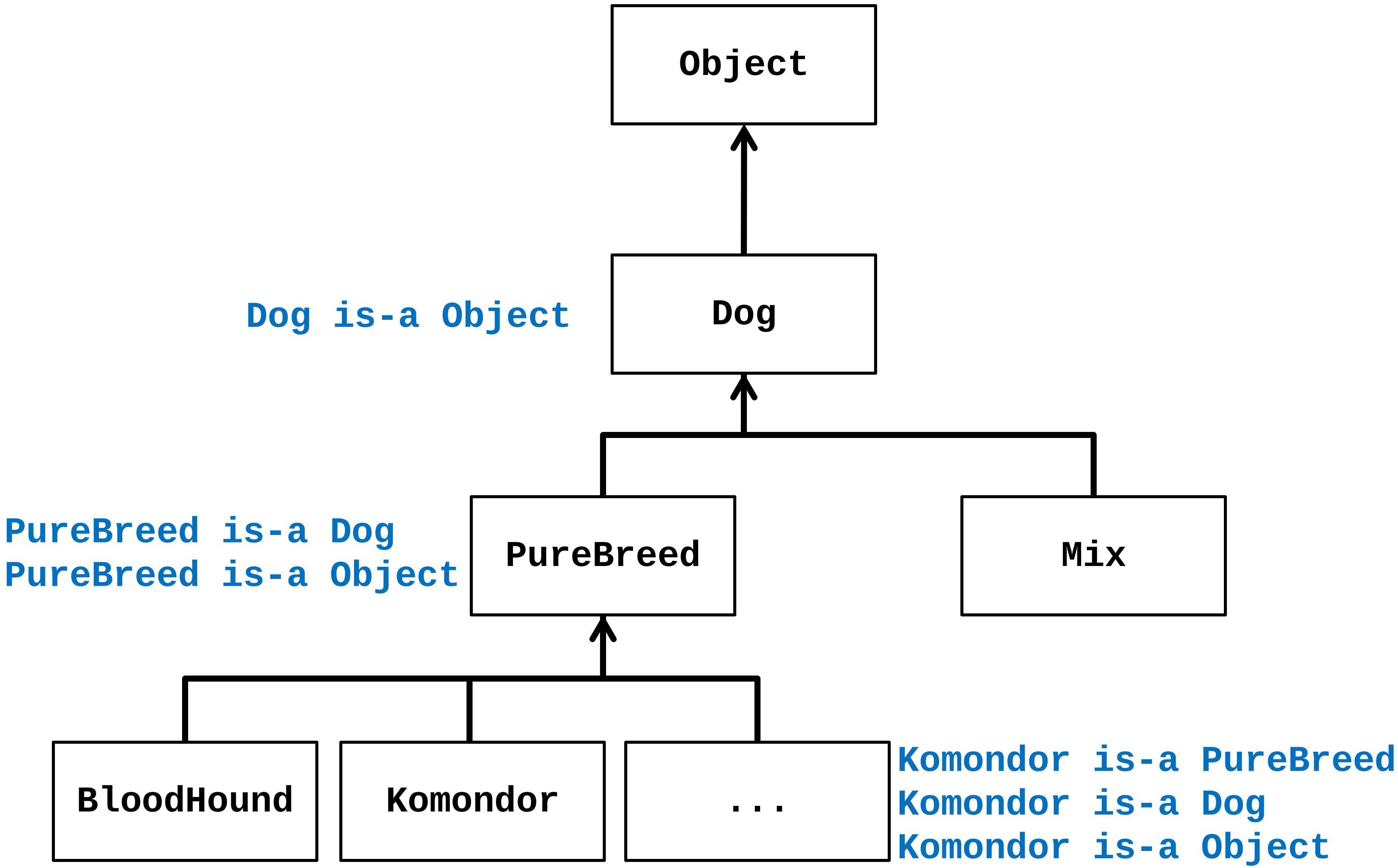
Inheritance

- means

is-a

or

is-substitutable-for

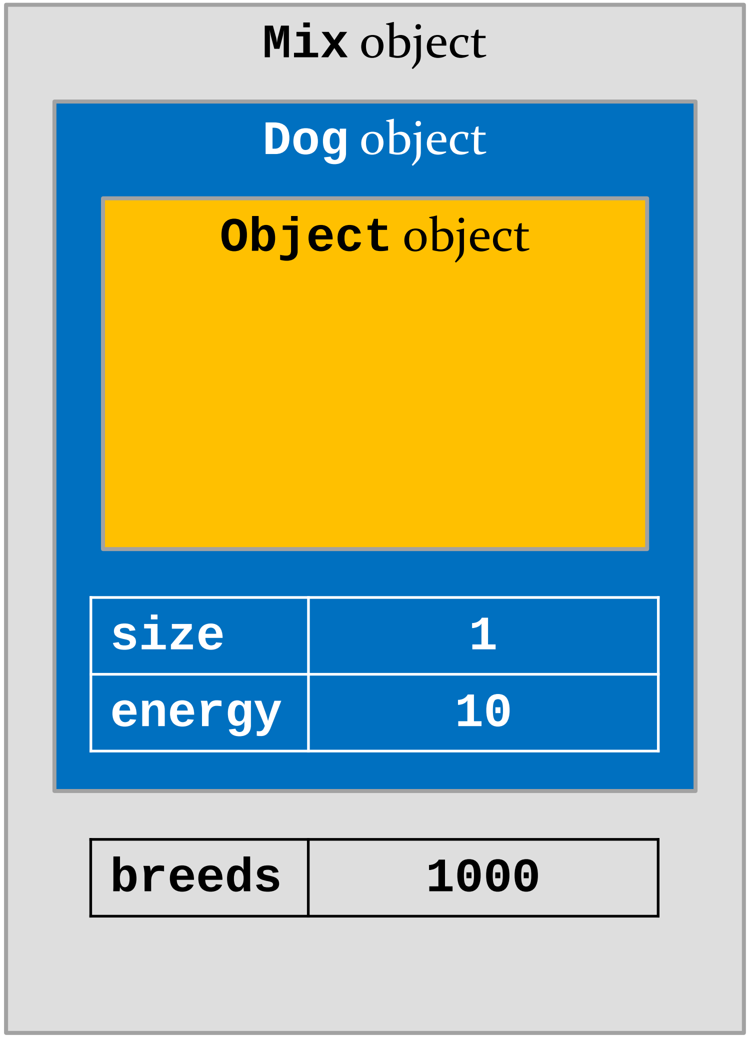

Object
Dog
Dog is-a Object
PureBreed is-a Dog
PureBreed is-a Object
PureBreed
Mix
BloodHound
Komondor
...
Komondor is-a PureBreed
Komondor is-a Dog
Komondor is-a Object

What is a Subclass?

- a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and attributes
- inheritance does more than copy the API of the superclass
  - the derived class contains a subobject of the parent class
  - the superclass subobject needs to be constructed (just like a regular object)
    - the mechanism to perform the construction of the superclass subobject is to call the superclass constructor

```
Mix mutt = new Mix(1, 10);
```

1. **Mix** constructor starts running
- creates new **Dog** subobject by invoking the **Dog** constructor
    2. **Dog** constructor starts running
    - creates new **Object** subobject by (silently) invoking the **Object** constructor
        3. **Object** constructor runs
    - sets **size** and **energy**
- creates a new empty **ArrayList** and assigns it to **breeds**



Strength of a Precondition

- to strengthen a precondition means to make the precondition more restrictive

```
// Dog setEnergy
// 1. no precondition
// 2. 1 <= energy
// 3. 1 <= energy <= 10
public void setEnergy(int energy)
{ ... }
```

weakest precondition

strongest precondition

Preconditions on Overridden Methods

- a subclass can change a precondition on a method *but it must not strengthen the precondition*
  - a subclass that strengthens a precondition is saying that it cannot do everything its superclass can do

```
// Dog setEnergy
// assume non-final
// @pre. none

public
void setEnergy(int nrg)
{ // ... }
```

```
// Mix setEnergy
// bad : strengthen precond.
// @pre. 1 <= nrg <= 10

public
void setEnergy(int nrg)
{
  if (nrg < 1 || nrg > 10)
  { // throws exception }
  // ...
}
```

- client code written for **Dog**s now fails when given a **Mix**

```
// client code that sets a Dog's energy to zero
public void walk(Dog d)
{
  d.setEnergy(0);
}
```

- remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

Strength of a Postcondition

- to strengthen a postcondition means to make the postcondition more restrictive

```
// Dog getSize
// 1. no postcondition
// 2. 1 <= this.size
// 3. 1 <= this.size <= 10
public int getSize()
{ ... }
```

weakest postcondition

strongest postcondition

Postconditions on Overridden Methods

- a subclass can change a postcondition on a method *but it must not weaken the postcondition*
  - a subclass that weakens a postcondition is saying that it cannot do everything its superclass can do

```
// Dog getSize
//
// @post. 1 <= size <= 10

public
int getSize()
{ // ... }
```

```
// Dogzilla getSize
// bad : weaken postcond.
// @post. 1 <= size

public
int getSize()
{ // ... }
```

Dogzilla: a made-up breed of dog that has no upper limit on its size

- client code written for **`Dog`**s can now fail when given a **`Dogzilla`**

```
// client code that assumes Dog size <= 10
public String sizeToString(Dog d)
{
  int sz = d.getSize();
  String result = "";
  if (sz < 4)         result = "small";
  else if (sz < 7)    result = "medium";
  else if (sz <= 10) result = "large";
  return result;
}
```

- remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

Exceptions and Inheritance

- a method that claims to throw an exception of type **`X`** is allowed to throw any exception type that is a subclass of **`X`**
  - this makes sense because exceptions are objects and subclass objects are substitutable for ancestor classes

```
// in Dog
public void someDogMethod() throws DogException
{
  // can throw a DogException, BadSizeException,
  //              NoFoodException, or BadDogException
}
```

- a method that overrides a superclass method that claims to throw an exception of type **X** must also throw an exception of type **X** or a subclass of **X**
  - remember: a subclass promises to do everything its superclass does; if the superclass method claims to throw an exception then the subclass must also

```
// in Mix
@Override
public void someDogMethod() throws DogException
{
  // ...
}
```

checked exception

Which are Legal?

- in Mix

```
@Override
public void someDogMethod() throws BadDogException

@Override
public void someDogMethod() throws Exception

@Override
public void someDogMethod()

@Override
public void someDogMethod()
        throws DogException, IllegalArgumentException
```

technically legal, but don't do this

Abstract Classes

- abstract classes appear when there are common attributes and methods that all subclasses share
- often, only the subclasses will have enough information to implement the methods
  - these methods are marked abstract in the parent class to indicate that subclasses are responsible for providing the implementation

Static Features and Inheritance

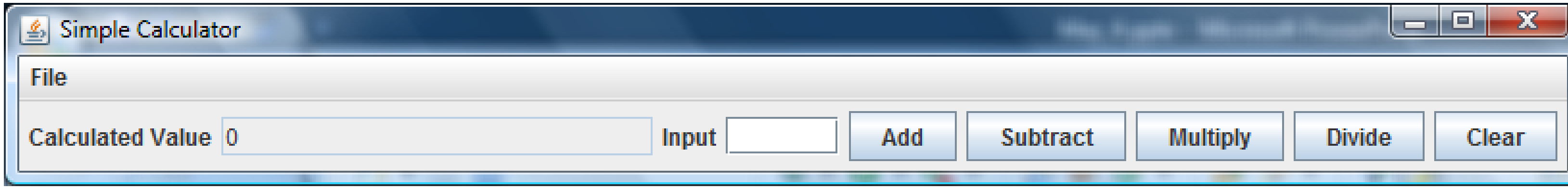
- non-private static attributes are inherited
  - but there is still only one copy of the attribute and it is in the parent class
- non-private static methods are inherited
  - but they cannot be overridden, they can only be hidden

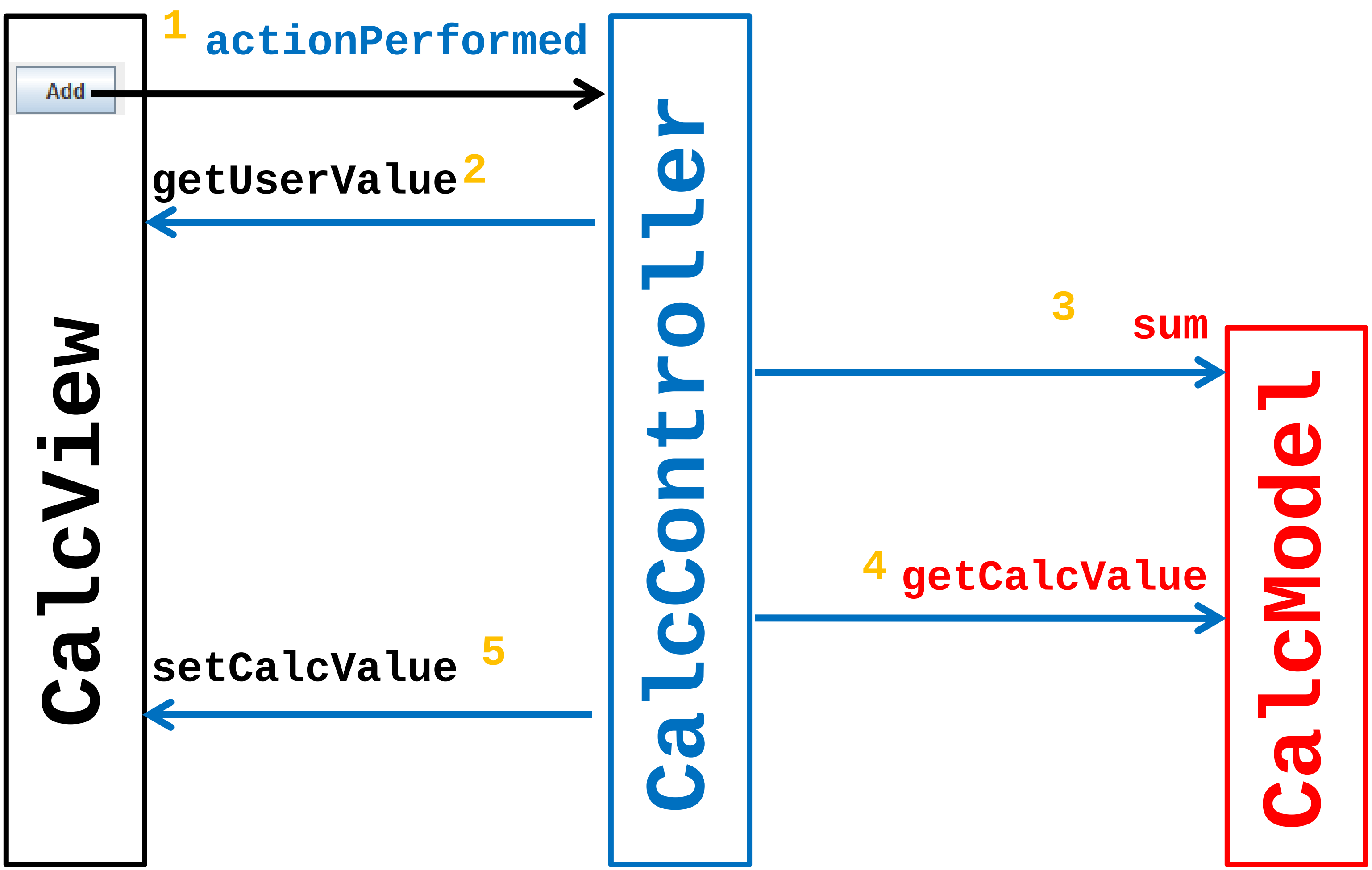
Interfaces

- in Java an *interface* is a reference type (similar to a class)
- an interface says what methods an object must have and what the methods are supposed to do
  - i.e., an interface is an API
- unlike inheritance, a class may implement as many interfaces as needed

Model-View-Controller

- model
  - represents state of the application and the rules that govern access to and updates of state
- view
  - presents the user with a sensory (visual, audio, haptic) representation of the model state
  - a user interface element (the user interface for simple applications)
- controller
  - processes and responds to events (such as user actions) from the view and translates them to model method calls


Simple Calculator
File
Calculated Value
0
Input
Add
Subtract
Multiply
Divide
Clear


CalcView
Add
1 actionPerformed
getUserValue 2
setCalcValue 5
CalcController
3 sum
4 getCalcValue
CalcModel

Recursion

- a method that calls itself is called a *recursive* method
- a recursive method solves a problem by repeatedly reducing the problem so that a base case can be reached

```
printIt("*", 5)
*printIt("*", 4)
**printIt("*", 3)
***printIt("*", 2)
****printIt("*", 1)
*****printIt("*", 0) base case
*****
```

Notice that the number of times the string is printed decreases after each recursive call to printIt

Notice that the base case is eventually reached.

Proving Correctness and Termination

- to show that a recursive method accomplishes its goal you must prove:
  1. that the base case(s) and the recursive calls are correct
  2. that the method terminates

Proving Correctness

- to prove correctness:
  1. prove that each base case is correct
  2. assume that the recursive invocation is correct and then prove that each recursive case is correct

Correctness of printItToo

1. (prove the base case) If `n == 0` nothing is printed; thus the base case is correct.
2. Assume that `printItToo(s, n-1)` prints the string `s` exactly `(n – 1)` times. Then the recursive case prints the string `s` exactly `(n – 1)+1 = n` times; thus the recursive case is correct.

Proving Termination

- to prove that a recursive method terminates:
  1. define the size of a method invocation; the size must be a non-negative integer number
  2. prove that each recursive invocation has a smaller size than the original invocation

Termination of printIt

1. **`printIt(s, n)`** prints **`n`** copies of the string **`s`**; define the size of **`printIt(s, n)`** to be **`n`**
2. The size of the recursive invocation **`printIt(s, n-1)`** is **`n-1`** (by definition) which is smaller than the original size **`n`**.

Recurrence Relation

- analyzing the runtime of an algorithm often leads to a recurrence relation $T(n)$, e.g.,
  - $T(n) = 2T(n / 2) + O(n)$
  - $T(n) = T(n - 1) + T(n - 2)$
- solving the recurrence can sometimes be done by substitution

Solving the Recurrence Relation

$$
\begin{aligned}
T(n) &\rightarrow 2T(n/2) + O(n) \\
&\approx 2T(n/2) + n \\
&= 2[\ 2T(n/4) + n/2\ ] + n \\
&= 4T(n/4) + 2n \\
&= 4[\ 2T(n/8) + n/4\ ] + 2n \\
&= 8T(n/8) + 3n \\
&= 8[\ 2T(n/16) + n/8\ ] + 3n \\
&= 16T(n/16) + 4n \\
&= 2^k T(n/2^k) + kn
\end{aligned}
$$

$T(n)$ approaches...

Solving the Recurrence Relation

$$T(n) = 2^k T(\underline{n/2^k}) + kn$$

- for a list of length $\mathbf{1}$ we know $T(\mathbf{1}) = \mathbf{1}$
  - if we can substitute $T(1)$ into the right-hand side of $T(n)$ we might be able to solve the recurrence

$$\underline{n/2^k} = \mathbf{1} \Rightarrow 2^k = n \Rightarrow k = \log(n)$$

Data Structures

- recursive
  - linked list
  - binary tree
- stack
- queue