Queues and their Applications

Lecture 24

Sections 19.4 - 19.6

Robb T. Koether

Hampden-Sydney College

Wed, Mar 21, 2018

Outline

1. Queues
2. The Queue Interface
3. Queue Applications
   - Infix Expression Evaluation
   - Bread-first Search
4. Assignment

Queues

Definition

A queue is a List that operates under the principle “first in, first out” (FIFO). New elements are enqueued into the queue. Old elements are dequeued from the queue.

- To enforce the FIFO principle, we enqueue and dequeue at opposite ends.

Implementation of Queues

- Which is more accurate?
  - A queue *is* a list.
  - A queue *has* a list.
- Use `pushFront()` and `popBack()`, or
- Use `pushBack()` and `popFront()`.
- Choose a List class for which enqueuing and dequeuing will be efficient.

Queue Implementation

- Choose an appropriate List class as a base class.
- Which are good choices?
  - `ArrayList`
  - `CircArrayList`
  - `LinkedList`
  - `LinkedListwTail`
  - `DoublyLinkedList`
  - `CircLinkedList`

Outline

Queue Constructors

Queue Constructors

```
Queue();
Queue(const Queue& q);
```

- `Queue()` constructs an empty queue.
- `Queue(Queue&)` constructs a copy of the specified queue.

Inspectors

Inspectors

```
T head() const;
int size() const;
bool isEmpty() const;
```

- `T head()` gets a copy of the element at the head of the queue.
- `int size()` gets the number of elements in the queue.
- `bool isEmpty()` determines whether the queue is empty.

Mutators

Mutators

```
void enqueue(const T& value);
T dequeue();
void makeEmpty();
```

- `enqueue()` enqueues the specified value at the tail of the queue.
- `dequeue()` dequeues and returns the element at the head of the queue.
- `makeEmpty()` makes the queue empty.

Facilitators

Facilitators

```
void input(istream& in);
void output(ostream& out) const;
```

- `input()` reads a queue from the specified input stream.
- `output()` writes a queue to the specified output stream.

Other Member Functions

Other Member Functions

```
void isValid() const;
```

- `isValid()` determines whether the queue has a valid structure.

Non-Member Functions

Non-Member Functions

```
istream& operator>>(istream& in, Queue& q);
ostream& operator<<(ostream& out, const Queue& q);
```

- **operator**>>() reads a queue from the specified input stream.
- **operator**<<() writes a queue to the specified output stream.

Input and Output

- Are there complications in using the List class `input()` and `output()` functions?
- Will the interpretation of head and tail be reversed between the `ArrayQueue` and the `LinkedQueue`?
- If so, then we might need to rewrite the functions for one of the two Queue classes.

Outline

Outline

Infix Expression Evaluation

- An infix expression with one (binary) operator is written in the order: left-operand, operator, right-operand.
- Example: $3 + 4$.

Disadvantages of Infix Notation

- Parentheses are often needed to indicate order of operation.

$$(3 + 4) * (5 + 6).$$

- Operators at different precedence levels follow a precedence hierarchy.

$$3 + 4 * 5 - 6/7.$$

- Operators at the same precedence level have a left or right associativity.

$$100 - 50 - 10 - 5 - 1.$$

Infix Expression Evaluation

Infix to Postfix Algorithm

- To evaluate an infix expression, we first convert it to postfix.
- Begin with an empty stack and an empty queue.
- Process the tokens from left to right according to the following rules.

Infix Expression Evaluation

Infix to Postfix Algorithm

- If the token is a number, enqueue the token.
- If the token is a left parenthesis, push the token onto the stack.
- If the token is a right parenthesis,
    - Pop tokens off the stack and enqueue them until a left parenthesis is popped.
    - Discard the right and left parentheses.

Infix Expression Evaluation

Infix to Postfix Algorithm

- If the token is an operator,
  - Pop tokens off the stack and enqueue them until
    - An operator of lower precedence is on top of the stack, or
    - A left parenthesis is on top of the stack, or
    - The stack is empty.
  - Push the operator onto the stack.

Infix Expression Evaluation

Infix to Postfix Algorithm

- After processing the last token, pop all tokens off the stack and enqueue them.
- The queue now contains the expression in post-fix notation.
- Now process the queue as a post-fix expression.

Example

- Use the algorithm to convert the expression

$$(7 + 5)/(9 - 3 * 2) * 6$$

to postfix.

Example

Token	Stack	Queue

Example

Token	Stack	Queue
(	(	

Example

Token	Stack	Queue
(	(	
7	(	7

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$
9	/ (	7 5 $+$ 9

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$
9	/ (	7 5 $+$ 9
$-$	/ ($-$	7 5 $+$ 9

Example

Token	Stack	Queue
(	(	
7	(	7
+	(+	7
5	(+	7 5
)		7 5 +
/	/	7 5 +
(	/ (	7 5 +
9	/ (	7 5 + 9
−	/ (−	7 5 + 9
3	/ (−	7 5 + 9 3

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$
9	/ (	7 5 $+$ 9
$-$	/ ($-$	7 5 $+$ 9
3	/ ($-$	7 5 $+$ 9 3
$*$	/ ($-$ $*$	7 5 $+$ 9 3

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$
9	/ (	7 5 $+$ 9
$-$	/ ($-$	7 5 $+$ 9
3	/ ($-$	7 5 $+$ 9 3
$*$	/ ($-$ $*$	7 5 $+$ 9 3
2	/ ($-$ $*$	7 5 $+$ 9 3 2

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$
9	/ (	7 5 $+$ 9
$-$	/ ($-$	7 5 $+$ 9
3	/ ($-$	7 5 $+$ 9 3
$*$	/ ($-$ $*$	7 5 $+$ 9 3
2	/ ($-$ $*$	7 5 $+$ 9 3 2
)	/	7 5 $+$ 9 3 2 $*$ $-$

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$
9	/ (	7 5 $+$ 9
$-$	/ ($-$	7 5 $+$ 9
3	/ ($-$	7 5 $+$ 9 3
$*$	/ ($-$ $*$	7 5 $+$ 9 3
2	/ ($-$ $*$	7 5 $+$ 9 3 2
)	/	7 5 $+$ 9 3 2 $*$ $-$
$*$	$*$	7 5 $+$ 9 3 2 $*$ $-$ /

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$
9	/ (	7 5 $+$ 9
$-$	/ ($-$	7 5 $+$ 9
3	/ ($-$	7 5 $+$ 9 3
$*$	/ ($-$ $*$	7 5 $+$ 9 3
2	/ ($-$ $*$	7 5 $+$ 9 3 2
)	/	7 5 $+$ 9 3 2 $*$ $-$
$*$	$*$	7 5 $+$ 9 3 2 $*$ $-$ /
6	$*$	7 5 $+$ 9 3 2 $*$ $-$ / 6

Example

Token	Stack	Queue
(	(	
7	(	7
$+$	($+$	7
5	($+$	7 5
)		7 5 $+$
/	/	7 5 $+$
(	/ (	7 5 $+$
9	/ (	7 5 $+$ 9
$-$	/ ($-$	7 5 $+$ 9
3	/ ($-$	7 5 $+$ 9 3
$*$	/ ($-$ $*$	7 5 $+$ 9 3
2	/ ($-$ $*$	7 5 $+$ 9 3 2
)	/	7 5 $+$ 9 3 2 $*$ $-$
$*$	$*$	7 5 $+$ 9 3 2 $*$ $-$ /
6	$*$	7 5 $+$ 9 3 2 $*$ $-$ / 6
(end)		7 5 $+$ 9 3 2 $*$ $-$ / 6 $*$

Outline

1. Queues
2. The Queue Interface
3. Queue Applications
   - Infix Expression Evaluation
   - Bread-first Search
4. Assignment

Tree Structures

- A tree is a structure that has the following properties.
  - It has a root node.
  - Every node may have any finite number of children nodes.
  - Every node except the root node has exactly one parent node.
  - There any “loops” in the tree.

Searching Trees

- Suppose that we have a tree structure that stores a value at each node.
- We would like to search the tree for a specific value.
- There are two general strategies:
  - Depth-first search
  - Breadth-first search

Depth-first Search

Definition

A depth-first search will follow a single path from parent to child until it reaches a dead-end. At that point, it backs up and follows a different path, and so on, until either it finds what it is looking for or it runs out of nodes to search.

- We would use a stack to implement a depth-first search.
  - We push each node we visit onto the stack.
  - When we hit a dead-end, we pop nodes from the stack until we can follow a different path.

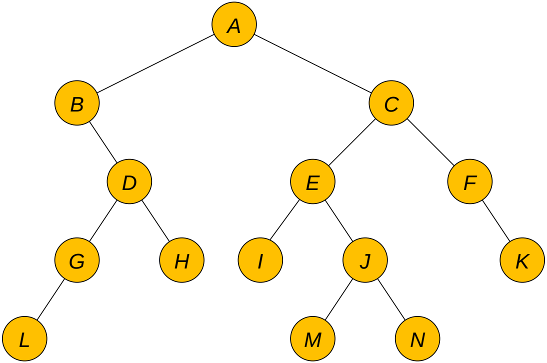
Breadth-first Search

Definition

A breadth-first search will visit all of the child nodes of the root node before it visits any of their children. It repeats this strategy level by level down the tree.

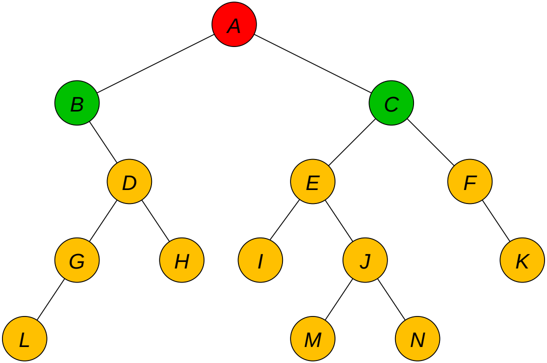
- We would use a queue to implement a breadth-first search.
  - Begin with the root node.
  - When a node is visited, enqueue all of its children.
  - Dequeue a node from the queue and visit that node next.

Breadth-first Search



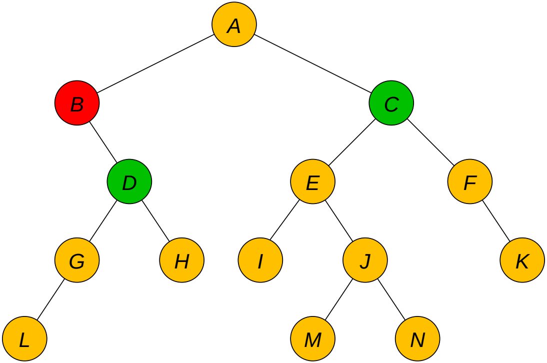
Queue:

Breadth-first Search



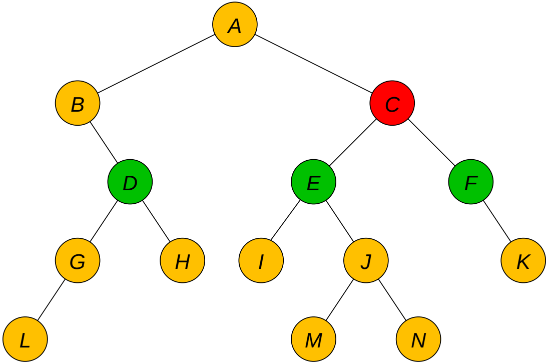
Queue: *B*, *C*

Breadth-first Search



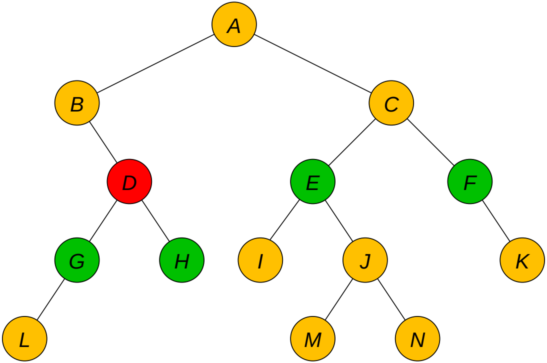
Queue: C, D

Breadth-first Search



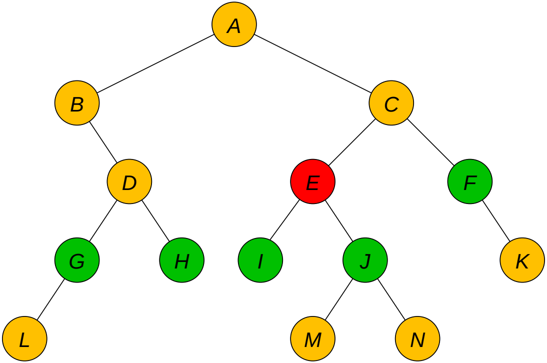
Queue: *D*, *E*, *F*

Breadth-first Search



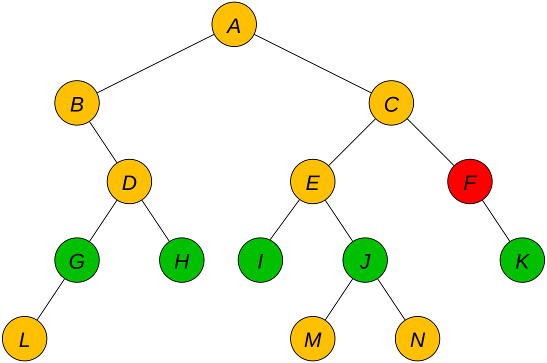
Queue: E, F, G, H

Breadth-first Search



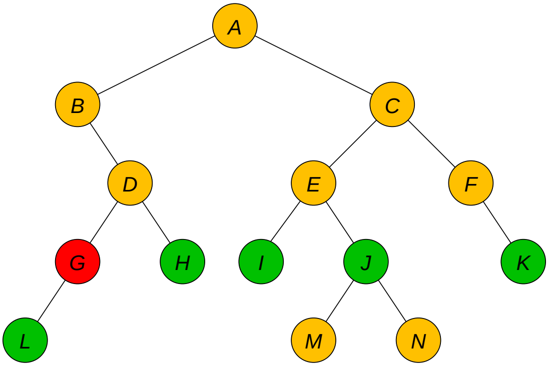
Queue: F, G, H, I, J

Breadth-first Search



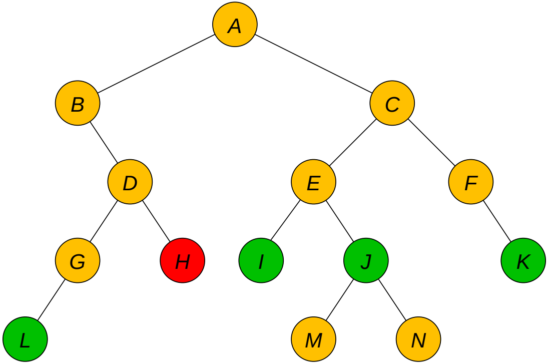
Queue: G, H, I, J, K

Breadth-first Search



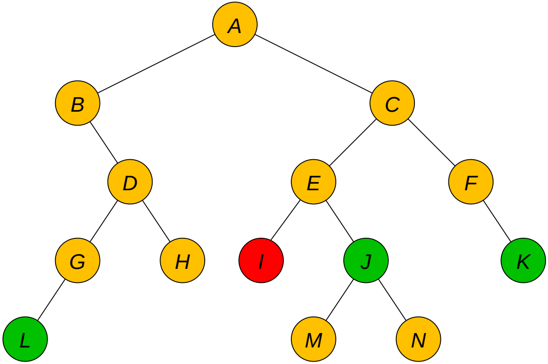
Queue: H, I, J, K, L

Breadth-first Search



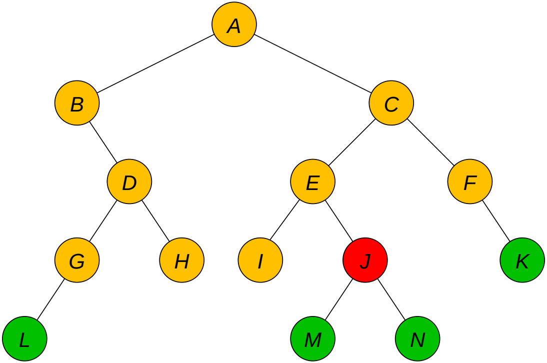
Queue: I, J, K, L

Breadth-first Search



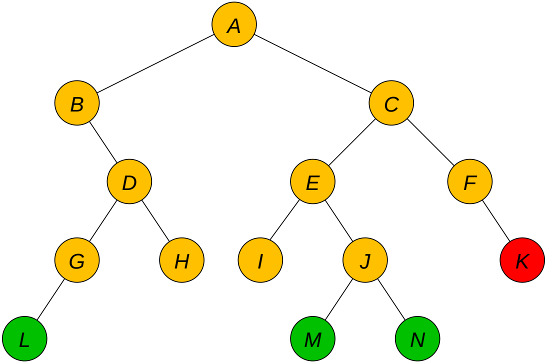
Queue: *J*, *K*, *L*

Breadth-first Search



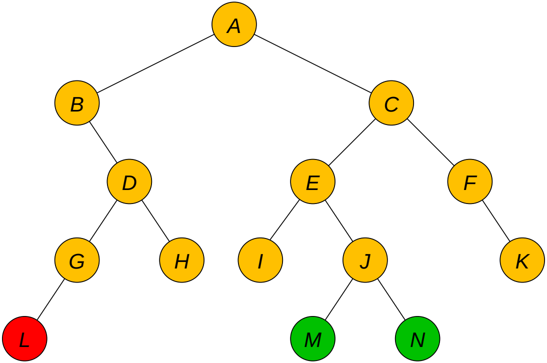
Queue: *K*, *L*, *M*, *N*

Breadth-first Search



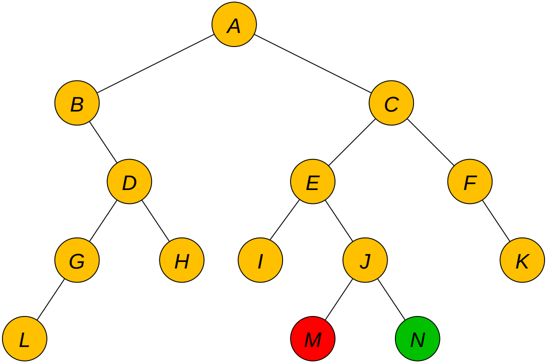
Queue: *L*, *M*, *N*

Breadth-first Search



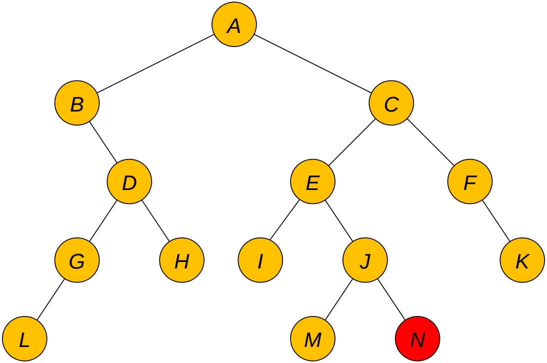
Queue: M, N

Breadth-first Search



Queue: *N*

Breadth-first Search



Queue:

Breadth-first Search

Current	Queue

Breadth-first Search

Current	Queue
A	B C

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$
D	$E\ F\ G\ H$

Breadth-first Search

Current	Queue
A	B C
B	C D
C	D E F
D	E F G H
E	F G H I J

Breadth-first Search

Current	Queue
A	B C
B	C D
C	D E F
D	E F G H
E	F G H I J
F	G H I J K

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$
D	$E\ F\ G\ H$
E	$F\ G\ H\ I\ J$
F	$G\ H\ I\ J\ K$
G	$H\ I\ J\ K\ L$

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$
D	$E\ F\ G\ H$
E	$F\ G\ H\ I\ J$
F	$G\ H\ I\ J\ K$
G	$H\ I\ J\ K\ L$
H	$I\ J\ K\ L$

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$
D	$E\ F\ G\ H$
E	$F\ G\ H\ I\ J$
F	$G\ H\ I\ J\ K$
G	$H\ I\ J\ K\ L$
H	$I\ J\ K\ L$
I	$J\ K\ L$

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$
D	$E\ F\ G\ H$
E	$F\ G\ H\ I\ J$
F	$G\ H\ I\ J\ K$
G	$H\ I\ J\ K\ L$
H	$I\ J\ K\ L$
I	$J\ K\ L$
J	$K\ L\ M\ N$

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$
D	$E\ F\ G\ H$
E	$F\ G\ H\ I\ J$
F	$G\ H\ I\ J\ K$
G	$H\ I\ J\ K\ L$
H	$I\ J\ K\ L$
I	$J\ K\ L$
J	$K\ L\ M\ N$
K	$L\ M\ N$

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$
D	$E\ F\ G\ H$
E	$F\ G\ H\ I\ J$
F	$G\ H\ I\ J\ K$
G	$H\ I\ J\ K\ L$
H	$I\ J\ K\ L$
I	$J\ K\ L$
J	$K\ L\ M\ N$
K	$L\ M\ N$
L	$M\ N$

Breadth-first Search

Current	Queue
A	$B\ C$
B	$C\ D$
C	$D\ E\ F$
D	$E\ F\ G\ H$
E	$F\ G\ H\ I\ J$
F	$G\ H\ I\ J\ K$
G	$H\ I\ J\ K\ L$
H	$I\ J\ K\ L$
I	$J\ K\ L$
J	$K\ L\ M\ N$
K	$L\ M\ N$
L	$M\ N$
M	N

Breadth-first Search

Current	Queue
A	B C
B	C D
C	D E F
D	E F G H
E	F G H I J
F	G H I J K
G	H I J K L
H	I J K L
I	J K L
J	K L M N
K	L M N
L	M N
M	N
N	

Outline

Assignment

Assignment

- Read Sections 19.4 - 19.6.