CSC 212
Data Structures and Abstractions
Fall 2015

Lecture 04: More on Linked Lists, Stacks, and Queues
Singly Linked List

5 → 7 → 2 → 6 (Tail)

Head

NULL pointer
if head == NULL
    throw exception
else if size == 1
    delete head
    head = tail = NULL
else
    p = head
    head = p->nxt
    delete p
if head == NULL
    throw exception
else if size == 1
    delete head
    head = tail = NULL
else
    p = head
    head = p->nxt
    delete p
if head == NULL
    throw exception
else if size == 1
    delete head
    head = tail = NULL
else
    p = head
    while p->nxt->nxt
        p = p->nxt
    tail = p
    delete tail->nxt
    tail->nxt = NULL
Delete at End

if head == NULL
    throw exception
else if size == 1
    delete head
    head = tail = NULL
else
    p = head
    while p->nxt->nxt
        p = p->nxt
    tail = p
    delete tail->nxt
    tail->nxt = NULL

O(n)
Delete by Value

\[ p = \text{head} \]
\[ \textbf{while} \ p \ \&\& \ p->\text{data} \neq \text{key} \]
\[ \quad q = p \]
\[ \quad p = p->\text{nxt} \]
\[ \textbf{if} \ \text{not} \ p \]
\[ \quad \text{throw exception} \]
\[ \textbf{else if} \ \text{head} == p \]
\[ \quad \text{call delete at front} \]
\[ \textbf{else if} \ \text{tail} == p \]
\[ \quad \text{call delete at end} \]
\[ \textbf{else} \]
\[ \quad q->\text{nxt} = p->\text{nxt} \]
\[ \quad \text{delete} \ p \]
Delete by Value

\[ p = \text{head} \]
\[ \text{while } p \land\land p->\text{data} \neq \text{key} \]
\[ q = p \]
\[ p = p->\text{nxt} \]
\[ \text{if not } p \]
\[ \text{throw exception} \]
\[ \text{else if } \text{head} == p \]
\[ \text{call delete at front} \]
\[ \text{else if } \text{tail} == p \]
\[ \text{call delete at end} \]
\[ \text{else} \]
\[ q->\text{nxt} = p->\text{nxt} \]
\[ \text{delete } p \]
Delete by Index

Try it yourself …

similar to delete by Value
Delete by Index

Try it yourself ...

similar to delete by Value

$O(n)$
Get

Try it yourself …

both Search by Value and by Index
Try it yourself …

both Search by Value and by Index

$O(n)$
Destroy

```c
p = head
while p
    q = p
    p = p->nxt
    delete q
head = tail = NULL
```
Destroy

\[ p = \text{head} \]

\[ \text{while } p \]

\[ q = p \]

\[ p = p->\text{nxt} \]

\[ \text{delete } q \]

\[ \text{head} = \text{tail} = \text{NULL} \]
Doubly Linked List

Head

5 → 7 → 2 → 6

Tail

NULL pointer
Circular Lists

**Tail** node points to **Head**

Can also have a [circular doubly linked list](#)
Stacks

LIFO Last In First Out
Stacks

LIFO Last In First Out

Push 5
Stacks

LIFO Last In First Out

Push 5
Push 3

3
5
Stacks

LIFO Last In First Out

Push 5
Push 3
Pop

5
Stacks

LIFO Last In First Out

Push 5
Push 3
Pop
Push 9

5
9
Stacks

Push 5
Push 3
Pop
Push 9
Push 8

LIFO Last In First Out
Stacks

Push 5
Push 3
Pop
Push 9
Push 8
Push 2

LIFO Last In First Out
Stacks

LIFO Last In First Out

Push 5
Push 3
Pop
Push 9
Push 8
Push 2
Pop

[8]
[9]
[5]
Stacks

LIFO Last In First Out
Stacks

LIFO Last In First Out
Stack Implementation

Arrays

can be fixed-size or dynamic array

push and pop at the end of array
Stack Implementation

Arrays can be fixed-size or **dynamic array**

**push** and **pop** at the end of array

\( O(1) \)

amortized cost for dynamic arrays
Stack Implementation

Arrays
- can be fixed-size or dynamic array
- push and pop at the end of array

Linked Lists
- push — insert at front
- pop — delete at front (needs to return value before deleting the node)

$O(1)$ amortized cost for dynamic arrays
Stack Implementation

Arrays
- can be fixed-size or **dynamic array**
- **push** and **pop** at the end of array

Linked Lists
- **push** — insert at front
- **pop** — delete at front (needs to return value before deleting the node)

\(O(1)\)

amortized cost for dynamic arrays
Considerations

Underflow

happens when \textbf{pop} from an empty stack
Considerations

Underflow
happens when \textit{pop} from an empty stack

Overflow
better use dynamic arrays or linked lists
Applications

Undo in software applications

Navigation buttons in browsers

Stack in compilers/programming languages

Parsing expressions

…
Example

Try with … fully parenthesized infix expressions

\[(5 + ((10 - 4) * (3+2)) + 25)\]

Two-Stack Algorithm [Dijkstra]

- value? push onto s1
- operator? push onto s2
- left parenthesis? ignore
- right parenthesis? pop two values from s1 and one operator from s2, apply operator and push resulting value to s1
Queues

FIFO First In First Out
Queues

Enqueue 5

FIFO First In First Out
Queues

Enqueue 5
Enqueue 3

FIFO First In First Out
Queues

Enqueue 5
Enqueue 3
Dequeue

FIFO First In First Out
Queues

Enqueue 5
Enqueue 3
Dequeue
Enqueue 9

FIFO First In First Out
Queues

FIFO First In First Out
Queues

Enqueue 5
Enqueue 3
Dequeue
Enqueue 9
Enqueue 8
Enqueue 2

FIFO First In First Out
Queues

Enqueue 5
Enqueue 3
Dequeue
Enqueue 9
Enqueue 8
Enqueue 2
Dequeue

FIFO First In First Out
Queues

Enqueue 5
Enqueue 3
Dequeue
Enqueue 9
Enqueue 8
Enqueue 2
Dequeue
Dequeue

FIFO First In First Out

8
2
Queues

FIFO First In First Out
Queue Implementation

Arrays

can be fixed-size or **dynamic array**

enqueue and dequeue (tricky to implement)
Queue Implementation

Arrays

can be fixed-size or **dynamic array**

enqueue and dequeue (tricky to implement)

Linked Lists

enqueue — insert at end

dequeue — delete at front (needs to return value before deleting the node)
Queue Implementation

Arrays

can be fixed-size or **dynamic array**
enqueue and dequeue (tricky to implement)

Linked Lists

enqueue — insert at end

dequeue — delete at front (needs to return value before deleting the node)
Queue Implementation

Arrays

- can be fixed-size or **dynamic array**
- **enqueue** and **dequeue** (tricky to implement)

Linked Lists

- **enqueue** — insert at end
- **dequeue** — delete at front (needs to return value before deleting the node)
Applications

Task/process scheduler

Media Playlists (Youtube, Spotify, Music, etc.)

Shared resource management

Simulations

Used in other algorithms

...