# 3. Basic Data Structures

# Stacks

## Abstrakter Datentyp Stack

- `new(S)`: neuer, leerer Stack `S`
- `isEmpty(S)`: boolean, ob `S` leer
- `pop(S)/pop()`: löscht oberstes Element von `S`, gibt es zurück; Fehler wenn `S` leer
- `push(S,k)/S.push(k)`: `k` als oberstes Element auf `S`; Fehler, wenn `S` voll
  LIFO: last in, first out

## Beispiel Bitcoin

#TODO Add

## Stacks als Array

- Annahme: maximale Größe `MAX` des Stacks vorher bekannt
- Zeiger `S.top` zeigt auf oberstes Element
- Zeiger wird bei Operationen passend bewegt

### Alogrithmen

```
new(S)
	S.A[]=ALLOCATE(MAX)
	S.top=-1;

isEmpty(S)
	IF S.top<0 THEN
		return true
	ELSE
		return false;

pop(S)
	IF isEmpty(S) THEN
		error 'underflow'
	ELSE
		S.top=S.top-1;
		return S.A[S.top+1];
```

```
push(S,k)
        IF S.top==MAX-1 THEN
                eror 'overflow'
        ELSE
                S.top=S.top+1;
                S.A[S.top]=k;
```

# Stacks variabler Größe

Wenn voll:

- Kopiere in größeres, zusammenhängendes Array oder
- Verteile auf viele Arrays, Siehe Verkettete Listen

## Einfache Lösung

Wenn voll, Array mit 1 Feld mehr erstellen, alles kopieren

### Laufzeit

Wenn $n$ Elemente in Array, $n$ `push`-Befehle führen zu $\Omega(n^2)$ Kopier-Schritten
Durchschnittlich $\Omega(n)$ Kopier-Schritte pro `push`

### Was tun

- Trivial: Unendlich viel speicher reservieren
- Gesucht: Lösung die maximal jeweils $O(\#\text{Elemente})$ braucht
  - Wenn Grenze erreicht, verdopple Speicher und kopiere um
  - Schrumpfe und kopiere, wenn weniger als $\frac{1}{4}$ benötigt

#### Algorithmen, Laufzeitanalyse

```
new(S)
        S.A[]=ALLOCATE(1);
        S.top=-1;
        S.memsize=1;


pop(S)
        IF isEmpty(S) THEN
                error 'underflow'
        ELSE
                S.top=S.top-1;
                IF 4*(S.top+1)==S.memsize THEN
```

```
                        S.memsize=S.memsize/2;
                        RESIZE(S.A,S.memsize);
                return S.A[S.top+1];

push(S,k)
        S.top=S.top+1;
        S.A[S.top]=k;
        IF S.top+1==S.memsize THEN
                S.memsize=2*S.memsize;
                RESIZE(S.A,S.memsize);
```

- `RESIZE(A,m)` reserviert neuen Speicher der Größe `m`, kopiert `A` um, fixt Referenz
- Im Schnitt für jeden der mindestens $n$ Befehle $\Theta(1)$ Umkopierschritte

# Verkettete Listen

## Datenstruktur doppelt verkettete Liste

- Element `x` besteht aus:
  - `key`: Wert
  - `prev`: Zeiger auf Vorgänger/`nil`
  - `nexxt`: Zeiger auf Nachfolger/`nil`
- `head` zeigt auf erstes Element (`nil` für leere Liste)

## Verkettete Listen durch Arrays

#TODO add image

## Elementare Operationen auf verketteten Listen

Im Pseudocode wird wie in Java von short circuit evaluation und call by reference/value verwendet

### Suche

```
search(L,k) // returns pointer to k in L (or nil)
        current=L.head;
        WHILE current != nil AND current.key != k DO // short circuit evaluation
                current=current.next;
        return current;
```

Laufzeit = $\Theta(n)$

## Einfügen

```
insert(L,x) // inserts element x in L
        x.next=L.head;
        x.prev=nil;
        IF L.head != nil THEN
                L.head.prev=x;
        L.head=x;
```

Laufzeit = $\Theta(1)$

- Prüft nicht, ob Wert bereits in Liste ist
- Wenn zuerst suche nach Wert stattfinden soll, $\Omega(n)$

## Löschen

```
delete(L,x) // deletes element x from L
IF x.prev != nil THEN
        x.prev.next=x.next
ELSE
        L.head=x.next;
IF x.next != nil THEN
        x.next.prev=x.prev;
```

Laufzeit = $\Theta(1)$

- `x` ist Verweis auf zu löschendes Element
- Wenn Wert gelöscht werden soll, muss dieser erst gesucht werden $\rightsquigarrow \Omega(n)$

# Vereinfachung per Wächter/Sentinels

- Ziel: eliminiere die Spezialfälle für Listenanfang/-ende
- Sentinel `L.sent` hinzugefügt, `head = L.sent.next`, `head.prev = L.sent`, `L.sent.key = nil`
- Für letztes Element `x` gilt: `x.next = L.sent` und `L.sent.prev = x`
- Sentinel ist von außen nicht sichtbar
- Leere Liste besteht nur aus Sentinel

## Löschen mit Sentinels

```
deleteSent(L,x) // deletes x from L with sentinel
        x.prev.next=x.next;
```

```
        x.next.prev=x.prev;
```

Andere Operationen müssen auch angepasst werden

# Queues

## Abstrakter Datentyp Queue

- `new(Q)`: Erzeugt neue, leere Queue namens `Q`
- `isEmpty(Q)`: Gibt an, ob `Q` leer
- `dequeue(Q)`: Gibt vorderstes Element aus `Q` zurückt, löscht es aus `Q`, Fehler wenn `Q` leer
- `enqueue(Q,k)`: Schreibt `k` als neues hinterstes Element auf `Q`, Fehler wenn `Q` voll
- FIFO: first in, first out

## Queues als virtuelles, zyklisches Array

- Problem mit Array-Implementierung:
  Queue "wandert", wenn Werte eingefügt/entfernt werden
- Führe `Q.rear, Q.front` für Zeiger auf Anfang und Ende ein
- Es gibt Ein Maximum für die Anzahl gleichzeitig in einer Queue: `MAX`
- Wenn `Q.rear, Q.front` auf selben Wert verweisen:
  - Speichere boolean `empty`, um anzugeben, ob Array vol oder leer
  - Alternativ: reserviere ein Element des Arrays als Abstandshalter

### Algorithmen

- `Q` leer, wenn `front==rear` und `empty==true`
- `Q` voll, wenn `front==rear` und `empty==false`

```
new(Q)
        Q.A[]=ALLOCATE(MAX);
        Q.front=0;
        Q.rear=0;
        Q.empty=true;

isEmpty(Q)
        return Q.empty;

dequeue(Q)
        IF isEmpty(Q) THEN
                error 'underflow'
```

```
        ELSE
                Q.front=Q.front+1 mod MAX;
                IF Q.front==Q.rear THEN
                        Q.empty=true;
                return Q.A[Q.front-1 mod MAX];

enqueue(Q,k)
        IF Q.rear==Q.front AND !Q.empty
        THEN error 'overflow'
        ELSE
                Q.A[Q.rear]=k;
                Q.rear=Q.rear+1 mod MAX;
                Q.empty=false;
```

## Queues durch einfach verkettete Listen

`front` und `rear` sind nun Zeiger auf Listenelemente

```
new(Q)
        Q.front=nil;
        Q.rear=nil;

isEmpty(Q)
        IF Q.front==nil THEN
                return true
        ELSE
                return false;

dequeue(Q)
        IF isEmpty(Q) THEN
                error 'underflow'
        ELSE
                x=Q.front;
                Q.front=Q.front.next;
                return x;

enqueue(Q,x)
        IF isEmpty(Q) THEN
                Q.front=x;
        ELSE
                Q.rear.next=x;
```

```
        x.next=nil;
        Q.rear=x;
```

# Anzahl Operationen Queues, Stacks, verkettete Listen

- Stack:
  - Push: $\Theta(1)$
  - Pop: $\Theta(1)$
- Queue:
  - Enqueue: $\Theta(1)$
  - Dequeue: $\Theta(1)$
- Verkettete Liste:
  - Einfügen: $\Theta(1)$
  - Löschen: $\Theta(1)$
  - Suchen: $\Theta(n)$
  - Löschen eines Wertes: $\Omega(n)$

# Binäre Bäume

## Bäume durch verkettete Listen

- `T.root` verweist auf Wurzelknoten des Baumes `T`
- Jeder Knoten enthält:
  - `key`: Wert
  - `child[]`: Array von Zeigern auf Kinder
  - manchmal auch `parent`: Zeiger auf Elternknoten
    Baum-Bedingung:
    Baum ist leer oder es gibt einen Knoten `r` (Wurzel), sodass jeder Knoten `v` von der Wurzel aus per eindeutiger Sequenz von `child`-Zeigern erreichbar ist:
    `v = r.child[i_1].child[i_2]. .....child[i_m]`

## Eigenschaften von Bäumen

- Bäume sind azyklisch
- Für nicht-leeren Baum gibt es genau $\#Knoten - 1$ viele Einträge $\neq nil$ über alle Listen `child[]`
- Man kann Bäume als (ungerichtete) Graphen darstellen, jedoch ist hier dann die Reihenfolge der Kinder relevant, da sie `child[]` abbilden muss

## Begrifflichkeiten

#TODO add image

# Binärbaum

Jeder Knoten hat maximal 2 Kinder: `left = child[0], right = child[1]`
Ausgangsgrad/outdegree jedes Knotens ist $\leq 2$
Markiere Knoten auch graphisch als linkes/rechtes Kind
Halbblatt: Knoten mit genau einem Kind
linker/rechter Teilbaum eine Knotens: Baum der links/rechts am Knoten hängt, d.h. der Baum, der den linken/rechten Kindknoten als Wurzel hat
Höhe des leeren Baumes ist -1
Höhe nicht-leeren Baumes = $\max\{\text{Höhe aller Teilbäume der Wurzel}\} + 1$

# Inorder-Traversieren von Binärbäumen

Beispielanwendung: Serialisierung

```
inorder(x)
        IF x != nil THEN
                inorder(x.left);
                print x.key;
                inorder(x.right);
```

Bei Bedarf mit Wrapper: `inoderTree(T) = inorder(T.root)`
$T(n)$ = Laufzeit bei $n$ Knoten, $T(n) \in O(n)$
Verschiedene Bäume können gleiche Inorder haben

# Pre- und Postorder-Traversieren von Binärbäumen

```
preorder(x)
        IF x != nil THEN
                print x.key;
                preorder(x.left);
                preorder(x.right);

postorder(x)
        IF x != nil THEN
                postorder(x.left);
                postorder(x.right);
                print x.key;
```

Preorder kann für Syntaxbäume bei funktionalen Programmiersprachen genutzt werden
Siehe auch #TODO Verweis auf Racket einfügen

### Preorder-Traversieren für Kopieren

1. Betrachte Knoten und lege Kopie an
2. Wiederhole die rekursive für Teilbäume

### Postorder-Traversieren für Löschen

1. Postorder löscht Teilbäume
2. Postorder betrachtet Knoten, an dem die Teilbäume hängen, erst danach, löscht zuletzt
   Verschiedene Bäume können gleiche Pre-/Postorder haben

## Binärbaum aus Preorder, Inorder und eindeutigen Werten

1. Preorder identifiziert Wurzel
2. Inorder identifiziert Werte im rechten/linken Teilbaum
3. Bilde Teilbäume rekursiv
   Statt Pre- auch Postorder möglich

## Abstrakter Datentyp Baum

- `new(T)`: Erzeugt neuen Baum `T`
- `search(T,k)`: Gibt Element `x` aus `T` mit `x.key==k` oder `nil` zurück
- `insert(T,x)`: Fügt `x` in `T` ein
- `delete(T,x)`: Löscht `x` aus `T`
  Oft gibt es weitere Baum-Operationen wie Wurzel, Höhe, Traversieren, ...

## Suchen

```
search(x,k)
        IF x==nil THEN return nil;
        IF x.key==k THEN return x;
        y=search(x.left,k);
        IF y != nil THEN return y;
        return search(x.right,k);
```

Starte mit `search(T.root,k)`
Laufzeit = $\Theta(n)$
Jeder Knoten wird maximal einmal besucht, im schlechtesten Fall aber auch jeder Knoten

## Einfügen

```
insert(T,x) // x.parent==x.left==x.right==nil;
        IF T.root != nil THEN
                T.root.parent=x;
                x.left=T.root;
        T.root=x;
```

Laufzeit = $\Theta(1)$
Erzeugt linkslastigen Baum

# Löschen

Idee: Ersetze `x` durch Halbblatt ganz rechts, es gibt auch andere Möglichkeiten
Sonderfälle beachten: Halbblatt hat selbst Wert `x` oder ist Wurzel

```
delete(T,x) // assumes x in T
        y=T.root;
        WHILE y.right!=nil DO
                y=y.right;
        connect(T,y,y.left);
        IF x != y THEN
                y.left=x.left;
                IF x.left != nil THEN
                        x.left.parent=y;
                y.right=x.right;
                IF x.right != nil THEN
                        x.right.parent=y;
                connect(T,x,y);

connect(T,y,w) // connects w to y.parent
        v=y.parent;
        IF y != T.root THEN
                IF y == v.right THEN
                        v.right=w;
                ELSE
                        v.left=w;
        ELSE
                T.root=w;
        IF w != nil THEN
                w.parent=v;
```

Bei `connect` muss `w` nicht an `y` hängen
Laufzeit `connect` = $\Theta(1)$

Laufzeit `delete` = $\Theta(h)$, $h$ ist Höhe des Baumes, $h = n$ ist möglich

# Binäre Suchbäume (Binary Search Tree, BST)

Wir nehmen totale Ordnung auf den Werten an
Binärer Suchbaum: Binärbaum, sodass für alle Knoten `z` gilt:

- Wenn `x` Knoten im linken Teilbaum von `z`, dann `x.key <= z.key`
- Wenn `y` Knoten im rechten Teilbaum von `z`, dann `y.key >= z.key`

## Order und eindeutige Werte

Aus Pre-/ Postorder und eindeutigen Werten kann man eindeutige BST konstruieren

1. Identifiziere Wurzel
2. Identifiziere Werte anhand der Regeln
3. Bilde Teilbäume rekursiv
   Mit Inorder und eindeutigen Werten lässt sich kein eindeutiger BST konstruieren

## Suche

```
search(x,k) // first call x=root
	IF x==nil OR x.key==k THEN
		return x;
	IF x.key > k THEN
		return search(x.left,k)
	ELSE
		return search(x.right,k);
```

Laufzeit $O(h)$, $h$ Höhe des Baumes

### Iterative Suche

```
iterative-search(x,k) // first call x=root
	WHILE x != nil AND x.key != k DO
		IF x.key > k THEN
			x=x.left
		ELSE
			x=x.right;
	return x;
```

## Einfügen

```
insert(T,z) // may insert z again, z.left==z.right==nil
        x=T.root; px=nil;
        WHILE x != nil DO
                px=x;
                IF x.key > z.key THEN
                        x=x.left
                ELSE
                        x=x.right;
        z.parent=px;
        IF px==nil THEN
                T.root=z
        ELSE
                IF px.key > z.key THEN
                        px.left=z
                ELSE
                        px.right=z;
```

Laufzeit $O(h)$

# Löschen

Zu löschender Knoten ist `z`, Fallunterscheidung:

- `z` hat maximal ein Kind:
  Kind anstelle von `z` setzen, fertig
  Wenn `z` Blatt ist, löschen trivial
  Bedingungen an Struktur/Werte bleiben erhalten
- Rechtes Kind von `z` hat kein linkes Kind:
  Analog: Linkes Kind von `z` hat kein rechtes Kind
  Rechtes Kind an die Stelle von `z` setzen, linkes Kind von `z` wird linkes Kind von rechtem Kind
  BST-Bedingung bleibt erhalten
- Kleinster Nachfahre vom rechten Kind von `z`:
  1. Finde kleinsten Nachfahren
  2. Ersetze `z` durch kleinsten Nachfahren
  3. Da kleinster Nachfahre kein linkes Kind haben kann, entsteht hier kein Problem
  4. Rechtes Kind des kleinsten Nachfahren an die Stelle des kleinsten Nachfahren

## Transplantation

hängt Teilbaum `v` an Elternknoten von `u`

```
transplant(T,u,v)
        IF u.parent==nil THEN
                T.root=v
        ELSE
                IF u==u.parent.left THEN
                        u.parent.left=v
                ELSE
                        u.parent.right=v;
        IF v != nil THEN
                v.parent=u.parent;
```

Laufzeit = $\Theta(1)$

## Algorithmus

```
delete(T,z)
        IF z.left==nil THEN
                transplant(T,z,z.right)
        ELSE
                IF z.right==nil THEN
                        transplant(T,z,z.left)
                ELSE
                        y=z.right;
                        WHILE y.left != nil DO y=y.left;
                        IF y.parent != z THEN
                                transplant(T,y,y.right);
                                y.right=z.right;
                                y.right.parent=y;
                        transplant(T,z,y);
                        y.left=z.left;
                        y.left.parent=y;
```

Laufzeit = $O(h)$

# Höhe des BST

## Laufzeit

Verkettete Liste:

- Einfügen: $\Theta(1)$
- Löschen: $\Theta(1)$

- Suchen: $\Theta(n)$
  BST:
- Einfügen: $O(h)$
- Löschen: $O(h)$
- Suchen: $O(h)$
  BST ist besser, wenn viele Such-Operationen durchgeführt werden und $h$ im Vergleich zu $n$ relativ klein ist

## Best-/Worst-Case

Best-Case:

- Vollständig: Alle Blätter haben gleiche Tiefe
- $h = O(\log_2 n)$
- Laufzeit = $O(\log_2 n)$
  Worst-Case:
- Degeneriert: Lineare Liste
- $h = n - 1$
- Laufzeit = $\Omega(n)$

## Durchschnittliche Höhe

Analyse ohne Einfügen und Löschen

```
randomlyBuiltTree(D) // D data set
        T=newTree();
        WHILE D != ∅ DO
                Pick d uniformly from D;
                insert(T,newNode(d));
                remove d from D;
        return T;
```

Die erwartete Höhe $E[h]$ des Baumes `T`, erzeugt durch `randomlyBuiltTree(D)`, für eine Datenmenge `D` mit $n$ Werten ist $E[h] = \Theta(\log_2 n)$.

# Suchbäume als Suchindex

- Knoten speichert nur Primärschlüssel und Zeiger auf Daten
- Bereichssuche ist möglich
- Sekundärindizes/zusätzliche Indizes kosten Speicherplatz und sind daher nur sinnvoll, wenn oft nach ihnen gesucht wird

- Z.B. sekundärer Baum mit alphabetischer Sortierung für eine Suche auf Namen