Informatik II

Trees, red-black trees
— SL06 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

DICTIONARIES/1

- Dictionaries store elements so that they can be located quickly using keys.
- Technically dictionaries manage (key, value) pairs. The value is often a pointer to the data.
- A dictionary may organize and manage bank accounts.
  - Each account is an object that is identified by an account number.
  - Each account stores a lot of additional information.
  - An application wishing to operate on an account would have to provide the account number as a search key.
  - Accesses without the search key (e.g., account number, personnel number) are poorly supported.

DICTIONARIES/1

- A dictionary D is a dynamic data structure with items that are (key, value) pairs and the following operations.
  - `search` (D, v): searches for v in D and, eventually, returns a pointer x such that $x \to key = v$
  - `delete` (D, x): deletes item pointed to by x from D
  - `insert` (D, x): inserts item pointed to by x into D

DICTIONARIES/1

- A dictionary D is a dynamic data structure with items that are (key, value) pairs and the following operations.
  - `search` (D, v): searches for v in D and, eventually, returns a pointer x such that $x \to key = v$
  - `delete` (D, x): deletes item pointed to by x from D
  - `insert` (D, x): inserts item pointed to by x into D
- Furthermore, one may want to support the following operations (which require that keys are comparable)
  - `minimum` (D) and `maximum` (D)
  - `successor` (D, v) and `predecessor` (D, v)

DICTIONARIES/2

Possible implementations

- Singly linked lists
  - Insert and delete: $\Theta(1)$
  - Search, min, max, succ, and pred: $\Theta(n)$
  - Ordered lists do not provide a real improvement

Dictionaries/2

Possible implementations

- Singly linked lists
  - Insert and delete: $\Theta(1)$
  - Search, min, max, succ, and pred: $\Theta(n)$
  - Ordered lists do not provide a real improvement
- Binary search trees
  - Represent search space as a binary tree
  - Search: $\Theta(\lg n)$ if tree is balanced
  - Insert & delete may yield unbalanced search trees

DICTIONARIES/2

Possible implementations

- Singly linked lists
  - Insert and delete: $\Theta(1)$
  - Search, min, max, succ, and pred: $\Theta(n)$
  - Ordered lists do not provide a real improvement
- Binary search trees
  - Represent search space as a binary tree
  - Search: $\Theta(\lg n)$ if tree is balanced
  - Insert & delete may yield unbalanced search trees
- Balanced binary trees
  - Cost of operations: $\Theta(\lg n)$
  - Red-black trees as device to keep trees balanced

TABLE OF CONTENTS — SL06

1. Binary trees
   - Binary trees
   - Binary search trees
     Inserting items
     Deleting items

2. Red-black trees

node has 2 children (max)
no order of values
no fast search

order to support fast search

Binary trees/1

Linked data structure

BINARY TREES/1

Linked data structure

- We assume that a node is accessed through a pointer to the node
- For each node x there is:
  - $x \rightarrow key$: value of the key (usually an integer value; not necessarily unique)
  - $x \rightarrow lft$: pointer to left child
  - $x \rightarrow rgt$: pointer to right child
  - other fields as needed

BINARY TREES/1

Linked data structure

- We assume that a node is accessed through a pointer to the node
- For each node x there is:
  - $x{\rightarrow}key$: value of the key (usually an integer value; not necessarily unique)
  - $x{\rightarrow}lft$: pointer to left child
  - $x{\rightarrow}rgt$: pointer to right child
  - other fields as needed
- $root$ points to the root of the tree

BINARY TREES/1

Linked data structure

- We assume that a node is accessed through a pointer to the node
- For each node x there is:
  - $x{\rightarrow}key$: value of the key (usually an integer value; not necessarily unique)
  - $x{\rightarrow}lft$: pointer to left child
  - $x{\rightarrow}rgt$: pointer to right child
  - other fields as needed
- $root$ points to the root of the tree
- For a leaf node p we have $p{\rightarrow}lft = p{\rightarrow}rgt = \texttt{NIL}$

BINARY TREES/2

Inorder tree walk

- Divide-and-conquer algorithm that visits all the nodes of a binary tree
- An inorder tree walk visits a node after visiting the left subtree and before visiting the right subtree.

BINARY TREES/2

Inorder tree walk

- Divide-and-conquer algorithm that visits all the nodes of a binary tree
- An inorder tree walk visits a node after visiting the left subtree and before visiting the right subtree.
- Running time of `InorderTreeWalk` is linear ($\Theta(n)$)

BINARY TREES/2

Inorder tree walk

- Divide-and-conquer algorithm that visits all the nodes of a binary tree
- An inorder tree walk visits a node after visiting the left subtree and before visiting the right subtree.
- Running time of `InorderTreeWalk` is linear ($\Theta(n)$)

The algorithm

Algo: InorderTreeWalk(p)

---

if $p \neq NIL$ **then**
  InorderTreeWalk(p->lft); - left
  VisitNode(p); print - node
  InorderTreeWalk(p->rgt); - right

Binary trees/3

Example tree walk

Binary trees/3

Example tree walk

`InorderTreeWalk` (3)

Binary trees/3

Example tree walk

`InorderTreeWalk` (3)
  `InorderTreeWalk` (1)

Binary trees/3

Example tree walk

```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
```

BINARY TREES/3

Example tree walk

`InorderTreeWalk` (3)
  `InorderTreeWalk` (1)
    `InorderTreeWalk` (`NIL`)
    `VisitNode` (1) — print 1

Binary trees/3

Example tree walk

`InorderTreeWalk` (3)
  `InorderTreeWalk` (1)
    `InorderTreeWalk` (`NIL`)
    `VisitNode` (1) — print 1
    `InorderTreeWalk` (4)

Binary trees/3

Example tree walk

```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
```

BINARY TREES/3

Example tree walk

```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
```

Binary trees/3

Example tree walk

```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
      InorderTreeWalk (NIL)
```

Binary trees/3

Example tree walk

```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
      InorderTreeWalk (NIL)
  VisitNode (3) — print 3
```

Binary trees/3

Example tree walk

```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
      InorderTreeWalk (NIL)
  VisitNode (3) — print 3
  InorderTreeWalk (6)
```

Binary trees/3

Example tree walk

```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
      InorderTreeWalk (NIL)
  VisitNode (3) — print 3
  InorderTreeWalk (6)
    [walk through right subtree]
```

BINARY TREES/4

- There are different ways to walk through a binary tree:
  - Inorder: visits a node after visiting the left subtree and before visiting the right subtree.
  - Preorder: visit a node before visiting its subtrees.
  - Postorder: visit a node after visiting its subtrees.
- To get these orders simply change the position of `VisitNode` (p) in the code of `InorderTreeWalk`.

EXERCISE SL06-1/1

6 - 3 - 4

(6-3) - 4 or ~~6 - (3 - 4)~~

Assume a binary tree is used to represent arithmetic expressions, such as $(5 + 6) * 7 - 3/8$. (((((5) + (6)) * (7)) - ((3) / (8))))

- Show the tree that corresponds to this expression.
- Given a tree print the expression in infix notation (operator between operands).
- Given a tree print the expression in postfix notation (operator after operands).

5 + 6 * 7 - 3 / 8

postfix

5 6 + 7 * 3 8 / -

EXERCISE SL06-1/2

```
infix(struct node* p)
    if (p ≠ NULL) then
        printf("(");
        infix(p->lft);
        printf("%c", p->sym);
        infix(p->rgt
        printf(")");
```

```
postfix(struct node* p)
    if (p ≠ NULL) then
        postfix(p->lft);
        postfix(p->rgt);
        printf("%c", p->sym);
```

```
struct node {
    char sym;
    struct node* lft;
    struct node* rgt;
}
```

Binary search trees/1

- A **binary search tree** is a binary tree where the nodes are ordered in a specific way.
- The organization supports searches for values.
- Binary search tree (and its variants) are used heavily since they allow to quickly find values in a large set of values.

BINARY SEARCH TREES/2

- Binary search tree property
  - Let x be a node in a binary tree
  - If y is a node in the left subtree of x then $y{\rightarrow}key \leq x{\rightarrow}key$
  - If y is a node in the right subtree of x then $x{\rightarrow}key \leq y.key$

Binary search trees/2

- Binary search tree property
  - Let x be a node in a binary tree
  - If y is a node in the left subtree of x then $y{\rightarrow}key \leq x{\rightarrow}key$
  - If y is a node in the right subtree of x then $x{\rightarrow}key \leq y.key$
- The keys of a binary search tree are stored such that they satisfy the above property

Binary search trees/2

- Binary search tree property
  - Let x be a node in a binary tree
  - If y is a node in the left subtree of x then $y{\rightarrow}key \leq x{\rightarrow}key$
  - If y is a node in the right subtree of x then $x{\rightarrow}key \leq y.key$
- The keys of a binary search tree are stored such that they satisfy the above property

BINARY SEARCH TREES/2

- Binary search tree property
  - Let x be a node in a binary tree
  - If y is a node in the left subtree of x then $y{\rightarrow}key \leq x{\rightarrow}key$
  - If y is a node in the right subtree of x then $x{\rightarrow}key \leq y.key$
- The keys of a binary search tree are stored such that they satisfy the above property

EXERCISE SL06-2

What is the difference between a binary search tree and a min heap? What is the time complexity to print the nodes of a tree with n nodes in sorted order with, respectively, a binary search tree and a min heap?

BST: $\forall$ nodes $x \in$ BST:

- keys in lft subtree of x are $\leq$ than x->key
- keys in rgt —— " — $\geq$ —— " —

MinHeap: $\forall$ nodes $x \in$ MinHeap:

- key in lft —— " — $\leq$ —— " ——
- rgt —— " — $\geq$

print in sorted order: BST: $\Theta(n)$ inorder tree walk

MinHeap: $\Theta(n \log n)$ – n times heapify

BINARY SEARCH TREES/3

Searching for an item

- To find an item with key $= v$ in a tree rooted at p
  - check whether p == `NIL` (v not in tree) or $p{\rightarrow}key = v$ (v was found)
  - if this is not the case then
    - if $v < p{\rightarrow}key$, search for v in $p{\rightarrow}lft$
    - otherwise, search for v in $p{\rightarrow}rgt$

BINARY SEARCH TREES/3

Searching for an item

- To find an item with key $= v$ in a tree rooted at p
  - check whether p == `NIL` (v not in tree) or $p{\rightarrow}key = v$ (v was found)
  - if this is not the case then
    - if $v < p{\rightarrow}key$, search for v in $p{\rightarrow}lft$
    - otherwise, search for v in $p{\rightarrow}rgt$
- Example: search for items with key $= 5$ and key $= 3$

BINARY SEARCH TREES/3

Searching for an item

- To find an item with key $= v$ in a tree rooted at p
  - check whether p == `NIL` (v not in tree) or $p{\rightarrow}key = v$ (v was found)
  - if this is not the case then
    - if $v < p{\rightarrow}key$, search for v in $p{\rightarrow}lft$
    - otherwise, search for v in $p{\rightarrow}rgt$
- Example: search for items with key $= 5$ and key $= 3$

BINARY SEARCH TREES/3

Searching for an item

- To find an item with key $= v$ in a tree rooted at p
  - check whether p == `NIL` (v not in tree) or $p{\rightarrow}key = v$ (v was found)
  - if this is not the case then
    - if $v < p{\rightarrow}key$, search for v in $p{\rightarrow}lft$
    - otherwise, search for v in $p{\rightarrow}rgt$
- Example: search for items with key $= 5$ and key $= 3$

BINARY SEARCH TREES/4

Iterative algorithm

Algo: TreeSearch(p,v)

---

while $p \neq NIL \wedge p\text{->}key \neq v$ **do**
  if $v < p\text{->}key$ **then** p = p->lft **else** p = p->rgt;

return p;

Binary search trees/4

Iterative algorithm

Algo: TreeSearch(p,v)

---

while $p \neq NIL \wedge p\text{->}key \neq v$ **do**
  if $v < p\text{->}key$ **then** p = p->lft **else** p = p->rgt;

return p;

Recursive algorithm

Algo: TreeSearch(p,v)

---

if $p == NIL \vee p\text{->}key == v$ **then return** p;
if $v < p\text{->}key$ **then return** TreeSearch(p->lft,v)
else return TreeSearch(p->rgt,v);

Binary search trees/5

Analysis of tree search

- Running time on tree of height h is $O(h)$

Binary search trees/5

Analysis of tree search

- Running time on tree of height h is $O(h)$
- In the worst case h is the number of nodes

Binary search trees/5

Analysis of tree search

- Running time on tree of height h is $O(h)$
- In the worst case h is the number of nodes

Finding the minimum

- Finds the minimum key in a tree rooted at p

Binary search trees/5

Analysis of tree search

- Running time on tree of height h is $O(h)$
- In the worst case h is the number of nodes

Finding the minimum

- Finds the minimum key in a tree rooted at p

Algo: TreeMin(p)

---

while *p->lft≠NIL* **do** p = p->lft;
return p;

BINARY SEARCH TREES/5

Analysis of tree search

- Running time on tree of height h is $O(h)$
- In the worst case h is the number of nodes

Finding the minimum

- Finds the minimum key in a tree rooted at p

Algo: TreeMin(p)

---

while *p->lft≠NIL* **do** p = p->lft;
return p;

- Running time is proportional to height of tree (can be $h \approx n$)
- We do not have to compare keys.

EXERCISE SL06-3

A function that counts the number of occurrences of an integer x in a binary search tree.

```
int cntX(struct node* p, int x)
  if (p ≠ NIL) then
    if (x < p->val) then return cntX(p->lft, x)
    else if (x > p->val) then return cntX(p->rgt, x)
    else return cntX(p->lft, x)
                +
                cntX(p->rgt, x)
                +
                1
```

Binary search trees/6

Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$

BINARY SEARCH TREES/6

Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$
- Distinguish two cases, depending on right subtree of p

BINARY SEARCH TREES/6

Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$
- Distinguish two cases, depending on right subtree of p

`TreeSucc` of 5 = 6

`TreeSucc` of 4 = 5

BINARY SEARCH TREES/6

Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$
- Distinguish two cases, depending on right subtree of p

`TreeSucc` of 5 = 6

`TreeSucc` of 4 = 5

- Case 1 ($p{\rightarrow}rgt \neq$ `NIL`)
  - successor of p is left-most node in right subtree

BINARY SEARCH TREES/6

Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$
- Distinguish two cases, depending on right subtree of p

`TreeSucc` of 5 = 6

`TreeSucc` of 4 = 5

- Case 1 ($p{\rightarrow}rgt \neq$ `NIL`)
  - successor of p is left-most node in right subtree
- Case 2 ($p{\rightarrow}rgt =$ `NIL`)
  - successor of p is lowest ancestor whose left subtree contains p

BINARY SEARCH TREES/7

- For (standard) binary trees without a parent pointer we need a pointer that lags behind and is only advanced if we move left.

```
struct node* SuccAbove(struct node* p,
                       struct node* x) {
  struct node* succ;
  succ = NULL;
  while (p != x) {
    if (x->key < p->key) { succ = p; p = p->lft; }
    else if (x->key > p->key) { p = p->rgt; }
  }
  return succ;
}
```

Usage: `SuccAbove(root, Search(root,v))`

Inserting items/1

Assumption and idea:

- Assume that a node p whose left and right children are NIL is given

Inserting items/1

Assumption and idea:

- Assume that a node p whose left and right children are NIL is given
- To insert the node p into a tree rooted at $root$
  - find location in tree where p belongs to
  - then add the new node p there
- Thus, insertion is similar to searching

Inserting items/1

Assumption and idea:

- Assume that a node p whose left and right children are NIL is given
- To insert the node p into a tree rooted at $root$
  - find location in tree where p belongs to
  - then add the new node p there
- Thus, insertion is similar to searching

Inserting items/1

Assumption and idea:

- Assume that a node p whose left and right children are NIL is given
- To insert the node p into a tree rooted at $root$
  - find location in tree where p belongs to
  - then add the new node p there
- Thus, insertion is similar to searching
- Makes use of a "one step delayed" pointer

Inserting items/1

Assumption and idea:

- Assume that a node p whose left and right children are NIL is given
- To insert the node p into a tree rooted at $root$
  - find location in tree where p belongs to
  - then add the new node p there
- Thus, insertion is similar to searching
- Makes use of a "one step delayed" pointer
- Running time on tree of height h is $O(h)$

Inserting items/2

Insertion example

Inserting items/2

Insertion example

Inserting items/2

Insertion example

Problematic instances

- If items are inserted in order, the tree looks like a list

Inserting items/2

Insertion example

Problematic instances

- If items are inserted in order, the tree looks like a list
- That is: height $h = n$, the number of nodes in the tree

Inserting items/2

Insertion example

Problematic instances

- If items are inserted in order, the tree looks like a list
- That is: height $h = n$, the number of nodes in the tree
- Worst case: the running time of the operations is $O(n)$

INSERTING ITEMS/3

Have a "one step delayed" pointer:

pointer to node that must be inserted

root of tree

```
struct node* TreeInsert(
  struct node* p, struct node* r)
{
  struct node* y = NULL;
  struct node* x = r;
  while (x != NULL) {
    y = x;
    if (x->key < p->key) x = x->rgt
    else x = x->lft;
  }
1) if (y == NULL) r = p;
2) else if (y->key < p->key) y->rgt = p;
3) else y->lft = p;
  return r;
}
```

we search for place where p must be inserted (x) and the node above (y)

call/use: root = TreeInsert(p, root)

EXERCISE SL06-4

Use representative trees to illustrate the insertion with a one step delayed pointer.

Deleting items/1

General remark

- Deletion depends on how many children a node has

Deleting items/1

General remark

- Deletion depends on how many children a node has

Deletion: case 1

- Node t has no children: remove t

Deleting items/1

General remark

- Deletion depends on how many children a node has

Deletion: case 1

- Node t has no children: remove t

Deleting items/1

General remark

- Deletion depends on how many children a node has

Deletion: case 1

- Node t has no children: remove t

DELETING ITEMS/2

Deletion: case 2

- Node t has one child x: let parent of t point to x and then remove t

Deleting items/2

Deletion: case 2

- Node t has one child x: let parent of t point to x and then remove t

Deleting items/2

Deletion: case 2 delete 2

- Node t has one child x: let parent of t point to x and then remove t

DELETING ITEMS/3

Deletion: case 3

- Node t has two children: find largest (smallest) child s in left (right) subtree of t, then replace $p{\rightarrow}key$ with $s.key$ and remove s

DELETING ITEMS/3

Deletion: case 3

- Node t has two children: find largest (smallest) child s in left (right) subtree of t, then replace $p{\rightarrow}key$ with $s.key$ and remove s

DELETING ITEMS/3

Deletion: case 3 delete 4

- Node t has two children: find largest (smallest) child s in left (right) subtree of t, then replace $\cancel{p}\rightarrow key$ t with $s.key$ and remove s

DELETING ITEMS/4

The algorithm

```
struct node* delete(struct node* root,
                    struct node* x) {

  u = root; v = NULL;
  while (u != x) {
    v = u;
    if (x->key < u->key) u = u->lft;
    else u = u->rgt;
  }

...
```

to be deleted

we search for x and make v above

DELETING ITEMS/5

If x has less than 2 children we fix the pointer of parent of x:

```
...

 if (u->rgt == NULL) {
1) if (v == NULL) root = u->lft;
2) else if (v->lft == u) v->lft = u->lft;
3) else v->rgt = u->lft;
 else if (u->lft == NULL) {
4) if (v == NULL) root = u->rgt;
5) else if (v->lft == u) v->lft = u->rgt;
6) else v->rgt = u->rgt;
 else {

...
```

use: p = TreeSearch(r, 17);
if (p ≠ NULL) { r = delete(r, p); }

DELETING ITEMS/6

If x has 2 children:

one node above p

special case (p=q):

largest node in left subtree of x

```
...
  p = x->lft; q = p;
  while (p->rgt != NULL) { q = p; p = p->rgt; }

  if (v == NULL) root = p;
  else if (v->lft == u) v->lft = p;
  else v->rgt = p;

  p->rgt = u->rgt;
  if (q != p) {
    q->rgt = p->lft;
    p->lft = u->lft;
  }
  return root
}
```

EXERCISE SL06-5

call: h = height(root, 0)

Given a binary tree write a function that determines its height.

```
int height(struct node * p, int h)
    if (p == NULL) return h-1;
    return max(height(p->lft, h+1),
               height(p->rgt, h+1));
```

h = 4

h = -1

h = 1

h = 0

TABLE OF CONTENTS — SL06

1. Binary trees

2. Red-black trees
   Inserting items
   Deleting items

- goal: Balanced BST with a guarantee for worst case running time

- recipe (not complete code)

Red-black trees/1

Example: Java's TreeMap class uses red-black trees

java.time.format
java.time.temporal
java.time.zone
java.util
java.util.concurrent
java.util.concurrent.atomic
java.util.concurrent.locks
java.util.function
java.util.jar
java.util.logging
java.util.prefs
java.util.regex

Spliterators.AbstractLongSpliterator
Spliterators.AbstractSpliterator
SplittableRandom
Stack
StringJoiner
StringTokenizer
Timer
TimerTask
TimeZone
TreeMap
TreeSet
UUID
Vector
WeakHashMap

Enums

Formatter.BigDecimalLayoutForm
Locale.Category
Locale.FilteringMode

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.util

Class TreeMap<K,V>

java.lang.Object
  java.util.AbstractMap<K,V>
    java.util.TreeMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>

---

public class **TreeMap<K,V>**
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable

A Red-Black tree-based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

Red-black trees/2

Balanced binary trees

- Problem: running time of tree operations is $O(h)$, worst case is $O(n)$

Red-black trees/2

Balanced binary trees

- Problem: running time of tree operations is $O(h)$, worst case is $O(n)$

Red-black trees/2

Balanced binary trees

- Problem: running time of tree operations is $O(h)$, worst case is $O(n)$

- Solution: balanced search trees guarantee small height $h = O(\lg n)$

Red-black trees/2

Balanced binary trees

- Problem: running time of tree operations is $O(h)$, worst case is $O(n)$

- Solution: balanced search trees guarantee small height $h = O(\lg n)$

RED-BLACK TREES/3

why? enforce/guarantee roughly balanced trees. properties can be maintained efficiently when inserting and deleting nodes

Red-black property

- A red-black tree is a binary search tree with the following additional properties
  1. Every node is either red or black
  2. The root is black
  3. Every leaf (`NIL`) is black
  4. If a node is red then both its children are black
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

RED-BLACK TREES/3

Red-black property

- A red-black tree is a binary search tree with the following additional properties
  1. Every node is either red or black
  2. The root is black
  3. Every leaf (`NIL`) is black
  4. If a node is red then both its children are black
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes
- Black-height $bh(x)$ of node x: number of black nodes on any path from, but not including, x down to a leaf

RED-BLACK TREES/3

Red-black property

- A red-black tree is a binary search tree with the following additional properties
  1. Every node is either red or black
  2. The root is black
  3. Every leaf (`NIL`) is black
  4. If a node is red then both its children are black
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes
- Black-height $bh(x)$ of node x: number of black nodes on any path from, but not including, x down to a leaf
- Black-height bh of tree: $bh(root)$, black-height of root

Red-black trees/4

Consequences/1

Red-black trees/4

Consequences/1

- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height

Red-black trees/4

Consequences/1

- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$

Red-black trees/4

Consequences/1

- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$
- $2^{bh} - 1 \leq n$

Red-black trees/4

Consequences/1

- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$ max # of red nodes on a path
- $2^{bh} - 1 \leq n$ ← all nodes Black
- $2^{h/2} \leq n + 1$

Red-black trees/4

Consequences/1

- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$
- $2^{bh} - 1 \leq n$
- $2^{h/2} \leq n + 1$
- $h/2 \leq \lg(n + 1)$

Red-black trees/4

Consequences/1

- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$
- $2^{bh} - 1 \leq n$
- $2^{h/2} \leq n + 1$
- $h/2 \leq \lg(n+1)$
- $\boxed{h \leq 2\lg(n+1)}$

$\lg n$ — .. $2\lg n$

80 90

n

1 Million

Red-black trees/5

Consequences/2

- Operations on a binary search tree (search, insert, delete, etc.) are accomplished in $O(h)$ time

Red-black trees/5

Consequences/2

- Operations on a binary search tree (search, insert, delete, etc.) are accomplished in $O(h)$ time
- A RB-tree is a binary search tree whose height is bound by $2\lg(n+1)$

RED-BLACK TREES/5

Consequences/2

- Operations on a binary search tree (search, insert, delete, etc.) are accomplished in $O(h)$ time
- A RB-tree is a binary search tree whose height is bound by $2\lg(n+1)$
- Thus, operations are accomplished in $O(\lg n)$ time

RED-BLACK TREES/5

Consequences/2

- Operations on a binary search tree (search, insert, delete, etc.) are accomplished in $O(h)$ time
- A RB-tree is a binary search tree whose height is bound by $2\lg(n+1)$
- Thus, operations are accomplished in $O(\lg n)$ time
- Note that there is the following constraint
  - Insertions and deletions may destroy RB-property
  - So, cost for restoring it cannot be more than $O(h)$

Red-black trees/6

Left & right rotations/1

- To change pointer structure of tree

RED-BLACK TREES/6

rotation preserves ordering (to the left are smaller equal; to the right are larger equal)

Left & right rotations/1

- To change pointer structure of tree

order: $\alpha s \beta t \gamma$

$\texttt{RightRotate}(t) \Rightarrow$
$\Leftarrow \texttt{LeftRotate}(s)$

order: $\alpha s \beta t \gamma$

RED-BLACK TREES/6

Left & right rotations/1

- To change pointer structure of tree

- Applies to BSTs and maintains key ordering
  - $\forall a \in \alpha, b \in \beta, c \in \gamma :$
    $a{\rightarrow}key \leq s{\rightarrow}key \leq b{\rightarrow}key \leq t{\rightarrow}key \leq c{\rightarrow}key$

RED-BLACK TREES/6

Left & right rotations/1

- To change pointer structure of tree

- Applies to BSTs and maintains key ordering
  - $\forall a \in \alpha, b \in \beta, c \in \gamma :$
    $a{\rightarrow}key \leq s{\rightarrow}key \leq b{\rightarrow}key \leq t{\rightarrow}key \leq c{\rightarrow}key$
- Rotations decrease/increase depth of α, γ (but not β)

Red-black trees/7

Left & right rotations/2

- Algorithm for right rotation (left rotation is equivalent)

RED-BLACK TREES/7

Left & right rotations/2

- Algorithm for right rotation (left rotation is equivalent)

Algo: RightRotate(t)

```
s = t->lft;
t->lft = s->rgt; s->pnt = t->pnt;
if s->rgt≠NIL then (s->rgt)->pnt = t;
if t->pnt==NIL then
  | root = s
else
  | if t==(t->pnt)->rgt then
  |   | (t->pnt)->rgt = s
  | else
  |   | (t->pnt)->lft = s
s->rgt = t; t->pnt = s;
```

RED-BLACK TREES/7

Left & right rotations/2

why implementation with child and parent pointers?
↳ impl gets simpler
↳ faster

- Algorithm for right rotation (left rotation is equivalent)

Algo: RightRotate(t)

```
s = t->lft;
t->lft = s->rgt; s->pnt = t->pnt;
if s->rgt≠NIL then (s->rgt)->pnt = t;
if t->pnt==NIL then
  | root = s
else
  | if t==(t->pnt)->rgt then
  |   | (t->pnt)->rgt = s
  | else
  |   |_(t->pnt)->lft = s
  |_
s->rgt = t; t->pnt = s;
```

- Running time is $O(1)$

Inserting items/1

The algorithm

Algo: RBTInsert(t)

```
s = NIL;
x = root;
while x≠NIL do
    s = x;
    if t->key<x->key then x = x->lft else x = x->rgt;
t->pnt = s; t->col = red;
if s==NIL then
    root = t
else
    if t->key<s->key then s->lft = t else s->rgt = t;
InsertFixup(t)
```

inserting into a binary search tree + set color of new node to red

Inserting items/2

- After an insertion some red-black properties might be violated. InsertFixup fixes any potential violations.

Setup:

- Conventions
  - t — new node
  - p — parent of t
  - g — grandfather of t
  - u — uncle of t
- Assumption
  - p is left child of g

Inserting items/2

- After an insertion some red-black properties might be violated. InsertFixup fixes any potential violations.

Setup:

- Conventions
  - t — new node
  - p — parent of t
  - g — grandfather of t
  - u — uncle of t
- Assumption
  - p is left child of g

Base case:

- If p is black no property is violated.

Inserting items/3

Insertion: case 1

- Condition: p is red, u is red

Inserting items/3

Insertion: case 1

- Condition: p is red, u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$

Inserting items/3

Insertion: case 1

- Condition: p is red, u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$

Inserting items/3

Insertion: case 1

- Condition: p is red, u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$

Inserting items/3

Insertion: case 1

- Condition: p is red, u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$

- Note that the tree rooted at g is now balanced

Inserting items/3

Insertion: case 1

- Condition: p is red, u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$

- Note that the tree rooted at g is now balanced
- g is now red, so the RB-property may be violated (propagate upwards with t = g)

EXERCISE SL06-6

Analyze case 1 wrt the properties of red black trees.

1. we have only red and black nodes: OK
2. root is black: no (red might go all the way up)
3. leaves are black: OK
4. no adjacent red nodes: action fixes this
5. same # of black nodes on all paths: OK

color root black if it goes all the way up

Inserting items/4

Insertion: case 2

- Condition: p is red, u is black, t is right child

Inserting items/4

Insertion: case 2

- Condition: p is red, u is black, t is right child
- Action: `LeftRotate` (p), set $t = p$

Inserting items/4

Insertion: case 2

- Condition: p is red, u is black, t is right child
- Action: `LeftRotate` (p), set $t = p$

Inserting items/4

Insertion: case 2

- Condition: p is red, u is black, t is right child
- Action: `LeftRotate` (p), set $t = p$

Inserting items/4

Insertion: case 2

- Condition: p is red, u is black, t is right child
- Action: `LeftRotate` (p), set $t = p$

- Note that in this way one gets case 3 (see below)

Inserting items/5

Insertion: case 3

- Condition: p is red, u is black, t is left child

INSERTING ITEMS/5

Insertion: case 3

- Condition: p is red, u is black, t is left child
- Action: $p{\rightarrow}col = black$, $g{\rightarrow}col = red$, `RightRotate` (g)

Inserting items/5

Insertion: case 3

- Condition: p is red, u is black, t is left child
- Action: $p{\rightarrow}col = black$, $g{\rightarrow}col = red$, `RightRotate` (g)

Inserting items/5

Insertion: case 3

- Condition: p is red, u is black, t is left child
- Action: $p{\rightarrow}col = black$, $g{\rightarrow}col = red$, `RightRotate` (g)

INSERTING ITEMS/5

Insertion: case 3

- Condition: p is red, u is black, t is left child
- Action: $p{\rightarrow}col = black$, $g{\rightarrow}col = red$, `RightRotate` (g)

- Note that the tree rooted at p is now balanced

└ no adj red nodes
└ # of black nodes on all paths has not changed

Inserting items/6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (exchange "left" and "right")

Inserting items/6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (exchange "left" and "right")

Final observations

- If two red nodes are adjacent, one does either
  - restructure (with one or two rotations) and stop, or
  - propagate adjacent red nodes upwards in the tree

Inserting items/6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (exchange "left" and "right")

Final observations

- If two red nodes are adjacent, one does either
  - restructure (with one or two rotations) and stop, or
  - propagate adjacent red nodes upwards in the tree
- A restructuring takes constant time and is done at most once (it reorganizes an off-balanced part of the tree)

Inserting items/6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (exchange "left" and "right")

Final observations

- If two red nodes are adjacent, one does either
  - restructure (with one or two rotations) and stop, or
  - propagate adjacent red nodes upwards in the tree
- A restructuring takes constant time and is done at most once (it reorganizes an off-balanced part of the tree)
- The number of propagations is bound by $O(\lg n)$

Inserting items/6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (exchange "left" and "right")

Final observations

- If two red nodes are adjacent, one does either
  - restructure (with one or two rotations) and stop, or
  - propagate adjacent red nodes upwards in the tree
- A restructuring takes constant time and is done at most once (it reorganizes an off-balanced part of the tree)
- The number of propagations is bound by $O(\lg n)$
- The running time of insertion is thus $O(\lg n)$

INSERTION EXAMPLE/1

- p is black
- nothing to do

Inserting C:

Insertion example/2

Inserting U:

EXERCISE SL06-7/1

Insert 36 into the following red black tree:

EXERCISE SL06-7/2

1) p lft ch
p is r, u is r
case 1

p.c = bl
u.c = bl
g.c = r

2) p lft ch
p is r, u is bl,
t rgt ch
case 2

LR(p); t := p

r lft ch
p is r u is bl
t lft ch
case 3

p.c = bl g.c = r RR(g)

DELETING ITEMS/1

colors of nodes do not change

- We first apply binary search tree deletion:
  - We can easily delete a node u that has at least one `NIL` child.
  - If the key to be deleted is stored at a node u with two children we replace the key of u with the key of the largest node v of the left subtree of u (no colors are changed) and delete v instead.

original node

replacement node

- Deletion of node s (s is either u or v)
  - Delete s and replace it by it's child x
  - If color of s was red we are done.
  - If color of s was black we proceed according to the cases described below

DELETING ITEMS/2

- After a deletion of a black node some red-black properties might be violated. The following actions fixes these violations.

Setup:

- Conventions
  - p — parent of x
  - b — brother of x
  - m — left child of b
  - n — right child of b
- Assumption
  - p is left child of g

DELETING ITEMS/2

- After a deletion of a black node some red-black properties might be violated. The following actions fixes these violations.

Setup:

- Conventions
  - p — parent of x
  - b — brother of x
  - m — left child of b
  - n — right child of b
- Assumption
  - p is left child of g

Base cases:

- If s was red noting to do.
- If s was black and x is red then color x black.

Deleting items/3

Deletion: case 1

- Condition: s is black, x is black, b is red

DELETING ITEMS/3

Deletion: case 1

- Condition: s is black, x is black, b is red
- Action: $b{\rightarrow}col = black$, $p{\rightarrow}col = red$, `LeftRotate` (p)

Deleting items/3

Deletion: case 1

- Condition: s is black, x is black, b is red
- Action: $b{\rightarrow}col = black$, $p{\rightarrow}col = red$, `LeftRotate` (p)

DELETING ITEMS/3

Deletion: case 1

- Condition: s is black, x is black, b is red
- Action: $b{\rightarrow}col = black$, $p{\rightarrow}col = red$, `LeftRotate` (p)

DELETING ITEMS/3

Deletion: case 1

- Condition: s is black, x is black, b is red
- Action: $b{\rightarrow}col = black$, $p{\rightarrow}col = red$, `LeftRotate` (p)

- Note that one gets either case 2, 3, or 4 (see below)

so on with $b = m$

# of missing bl nodes on root-leaf path

DELETING ITEMS/4

Deletion: case 2

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is black, $b{\rightarrow}rgt{\rightarrow}col$ is black,

DELETING ITEMS/4

Deletion: case 2

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is black, $b{\rightarrow}rgt{\rightarrow}col$ is black,
- Action: $b{\rightarrow}col = red$, set $x = p$

DELETING ITEMS/4

Deletion: case 2

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is black, $b{\rightarrow}rgt{\rightarrow}col$ is black,
- Action: $b{\rightarrow}col = red$, set $x = p$

DELETING ITEMS/4

Deletion: case 2

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is black, $b{\rightarrow}rgt{\rightarrow}col$ is black,
- Action: $b{\rightarrow}col = red$, set $x = p$

DELETING ITEMS/4

Deletion: case 2

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is black, $b{\rightarrow}rgt{\rightarrow}col$ is black,
- Action: $b{\rightarrow}col = red$, set $x = p$

- Note that bh of both subtrees of p is reduced by one

DELETING ITEMS/4

Deletion: case 2

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is black, $b{\rightarrow}rgt{\rightarrow}col$ is black,
- Action: $b{\rightarrow}col = red$, set $x = p$

- Note that bh of both subtrees of p is reduced by one
- Stop deletion if color of p is red and set $p{\rightarrow}col = black$

DELETING ITEMS/5

Deletion: case 3

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is red, $b{\rightarrow}rgt{\rightarrow}col$ is black

DELETING ITEMS/5

Deletion: case 3

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is red, $b{\rightarrow}rgt{\rightarrow}col$ is black
- Action: $b{\rightarrow}lft{\rightarrow}col = black$, $b{\rightarrow}col = red$, `RightRotate` (b)

DELETING ITEMS/5

Deletion: case 3

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is red, $b{\rightarrow}rgt{\rightarrow}col$ is black
- Action: $b{\rightarrow}lft{\rightarrow}col = black$, $b{\rightarrow}col = red$, `RightRotate` (b)

Deleting items/5

Deletion: case 3

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is red, $b{\rightarrow}rgt{\rightarrow}col$ is black
- Action: $b{\rightarrow}lft{\rightarrow}col = black$, $b{\rightarrow}col = red$, `RightRotate` (b)

DELETING ITEMS/5

Deletion: case 3

- Condition: s is black, x is black, b is black, $b{\rightarrow}lft{\rightarrow}col$ is red, $b{\rightarrow}rgt{\rightarrow}col$ is black
- Action: $b{\rightarrow}lft{\rightarrow}col = black$, $b{\rightarrow}col = red$, `RightRotate` (b)

- Note that in this way one gets case 4 (see below)

go on with $e = m$

DELETING ITEMS/6

Deletion: case 4

- Condition: s is black, x is black, b is black, $b{\rightarrow}rgt{\rightarrow}col$ is red

DELETING ITEMS/6

Deletion: case 4

- Condition: s is black, x is black, b is black, $b{\rightarrow}rgt{\rightarrow}col$ is red
- Action: $b{\rightarrow}col = p{\rightarrow}col$, $p{\rightarrow}col = b{\rightarrow}rgt{\rightarrow}col = black$, `LeftRotate` (p)

DELETING ITEMS/6

Deletion: case 4

- Condition: s is black, x is black, b is black, $b{\rightarrow}rgt{\rightarrow}col$ is red
- Action: $b{\rightarrow}col = p{\rightarrow}col$, $p{\rightarrow}col = b{\rightarrow}rgt{\rightarrow}col = black$, `LeftRotate` (p)

DELETING ITEMS/6

Deletion: case 4

- Condition: s is black, x is black, b is black, $b{\rightarrow}rgt{\rightarrow}col$ is red
- Action: $b{\rightarrow}col = p{\rightarrow}col$, $p{\rightarrow}col = b{\rightarrow}rgt{\rightarrow}col = black$, `LeftRotate` (p)

DELETING ITEMS/6

Deletion: case 4

- Condition: s is black, x is black, b is black, $b{\rightarrow}rgt{\rightarrow}col$ is red
- Action: $b{\rightarrow}col = p{\rightarrow}col$, $p{\rightarrow}col = b{\rightarrow}rgt{\rightarrow}col = black$, `LeftRotate` (p)

- Note that the tree rooted at b is now balanced

DELETING ITEMS/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

DELETING ITEMS/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- Intuitive idea is to perform a color compensation
  - Find a red node nearby and change its color into black

Deleting items/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- Intuitive idea is to perform a color compensation
  - Find a red node nearby and change its color into black
- There are two cases: (a) restructuring, (b) recoloring
  - (a) resolves problem locally, (b) propagates it upwards

Deleting items/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- Intuitive idea is to perform a color compensation
  - Find a red node nearby and change its color into black
- There are two cases: (a) restructuring, (b) recoloring
  - (a) resolves problem locally, (b) propagates it upwards
- The number of recolorings is bound by $O(\lg n)$

Deleting items/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- Intuitive idea is to perform a color compensation
  - Find a red node nearby and change its color into black
- There are two cases: (a) restructuring, (b) recoloring
  - (a) resolves problem locally, (b) propagates it upwards
- The number of recolorings is bound by $O(\lg n)$
- The running time of deletion is thus $O(\lg n)$

Deletion example/1

Deleting 9:

- x rgt ch
- s is bl, x is bl, b is b, m is bl, u is bl
- case 2m
- b.c=r, x=p

Deletion example/2

Deleting 7:

- x rgt ch
- s is Bl, x is Bl, b is Bl, b->lft->ch is r (m)
- case 4m
- b.c = p.c p.c = Bl m.c = Bl RR(p)

EXERCISE SL06-8/1

Consider the following red-black tree.

has to be replaced by 28 (= largest value in left subtree)

Delete the node with key 32 and draw the resulting tree.

Exercise SL06-8/2

- x lft ch
- s is Bl
  x is Bl
  b is Bl
  w is r
  u is Bl
- case 3
- w.C = Bl
  b.C = r
  RR(b)

- x lft ch
- s is Bl
  x is Bl
  b is Bl
  u is r
- case 4
- b.C = pC
  p.C = Bl
  u.C = Bl
  LR(p)

SUMMARY/1

Binary trees

- In a binary tree each node has up to two children.
- A binary search tree is a binary tree where
  - in the left subtree of a node x all nodes have keys less or equal to the key of node x
  - in the right subtree of a node x all nodes have keys larger or equal to the key of node x
- Binary search trees offer $log(n)$ performance for search, insertion and deletion if the tree is balanced.
- In the worst case binary trees degenerate to lists with linear performance.

SUMMARY/2

Red-black trees

- Red-black trees are binary search trees that are balanced.
- Red-black trees provide $log(n)$ worst case guarantees for search, insertion and deletion.
- Red-black trees are very popular in practice because of the worst-case guarantee.
- There are alternative balanced trees (sometimes very close to red-black trees):
  - AA trees
  - 2-3 trees
  - 2–3–4 tree (also called a 2–4 tree)
  - AVL trees

Informatik II

Hash tables

— SL07 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

Table of contents — SL07

1. Hash tables
   Dictionaries
   Hash tables

2. Hash functions

3. Collision resolution

Dictionaries revisited/1

colors
pictures

- Some dictionaries only require to check for membership (i.e., is element in dictionary and where?)
- If order (methods such as min, max, successor, predecessor) is not required it is enough to check for equality.
- Operations that require ordering are still possible (e.g., scanning the entire database) but cannot use the dictionary access structure.
  - Usually all elements must be compared, which is slow.
  - Can be OK if it is rare enough.

DICTIONARIES REVISITED/2

- AT&T is a large phone company, and they want to provide caller ID capability:
  - given a phone number, return the caller's name
  - phone numbers range from 0 to $r = 10^8 - 1$
  - want to do this as efficiently as possible
- Examples where usually no ordering is required:
  - AppleID
  - Insurance number
  - Phone number
  - Residential addresses
- Examples where ordering could be relevant:
  - IP addresses
  - ISBN number
  - S-Number
  - Names

DICTIONARIES REVISITED/3

- BST: insert, search, and delete in $O(h)$ time

DICTIONARIES REVISITED/3

- BST: insert, search, and delete in $O(h)$ time
- RBT: insert, search, and delete in $O(\lg n)$ time

DICTIONARIES REVISITED/3

- BST: insert, search, and delete in $O(h)$ time
- RBT: insert, search, and delete in $O(\lg n)$ time
- Hashing: insert, search, and delete in $O(1)$ time
  - Other dictionary operations (min, succ, ...) are not supported
  - Hash table is expensive and difficult to grow and shrink.
  - Application: symbol table for identifiers in a compiler, customer IDs, ...

hash tables are not good for very dynamic environments

Dictionaries revisited/3

- BST: insert, search, and delete in $O(h)$ time
- RBT: insert, search, and delete in $O(\lg n)$ time
- Hashing: insert, search, and delete in $O(1)$ time
  - Other dictionary operations (min, succ, ...) are not supported
  - Hash table is expensive and difficult to grow and shrink.
  - Application: symbol table for identifiers in a compiler, customer IDs, ...

Hash tables vs arrays

- Hash tables are a generalization of arrays

DICTIONARIES REVISITED/3

- BST: insert, search, and delete in $O(h)$ time
- RBT: insert, search, and delete in $O(\lg n)$ time
- Hashing: insert, search, and delete in $O(1)$ time
  - Other dictionary operations (min, succ, ...) are not supported
  - Hash table is expensive and difficult to grow and shrink.
  - Application: symbol table for identifiers in a compiler, customer IDs, ...

Hash tables vs arrays

- Hash tables are a generalization of arrays
- Big advantage of arrays consists in direct addressing
  - Ability to access arbitrary array position in $O(1)$ time

e.g. a[51749]

DICTIONARIES REVISITED/4

- In Java: java.util Map interface defines dictionary ADT
- Different data structures to realize dictionaries:
  - arrays
  - linked lists
  - Binary search trees
  - Red/Black trees
  - Hash tables
  - B-trees

DICTIONARIES REVISITED/5

Two suboptimal ways to design this dictionary

- direct addressing: an array indexed by key

(null)	(null)	...	Jens	...	(null)
0000-0000	0000-0001	...	9635-8904	...	9999-9999

  - Requires O(1) time, – amazing (searching)
  - Requires O(r) space - huge amount of wasted space
    – bad (a huge array)

- a linked list:

  - Requires O(n) time and O(n) space

Hash tables/1

- We use a **hash table** of size m to avoid wasting space.
- A hash table is like an array, but with a function to map the large range into one that we can manage efficiently.
- For instance take the original key, modulo the size of the relatively small hash table, and use that as an index.
- Example: insert *(9635-8904, Jens)* into a hash table with five slots ($m = 5$):
  - $96358904 \mod 5 = 4$

(null)	(null)	(null)	(null)	Jens
0	1	2	3	4

Hash tables/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k

Hash tables/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot

HASH TABLES/2

often use HT[0..m-1]
(because of modulo
Ref)

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot
- Maps universe U into the slots of hash table $HT[1 \ldots m]$

$$h : U \rightarrow \{1, 2, \ldots, m\}$$

item with key k hashes to $h(k)$, $h(k)$ is hash value of k

Hash tables/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot
- Maps universe U into the slots of hash table $HT[1 \ldots m]$

$$h\colon U \to \{1, 2, \ldots, m\}$$

  item with key k hashes to $h(k)$, $h(k)$ is hash value of k
- Hash function reduces range of array indices that need to be handled, i.e., m values instead of $|U|$ values

Hash tables/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot
- Maps universe U into the slots of hash table $HT[1 \ldots m]$

$$h\colon U \to \{1, 2, \ldots, m\}$$

  item with key k hashes to $h(k)$, $h(k)$ is hash value of k
- Hash function reduces range of array indices that need to be handled, i.e., m values instead of $|U|$ values
- There is a hitch: two keys may hash to the same slot

Hash tables/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot
- Maps universe U into the slots of hash table $HT[1 \ldots m]$

$$h\colon U \to \{1, 2, \ldots, m\}$$

  item with key k hashes to $h(k)$, $h(k)$ is hash value of k
- Hash function reduces range of array indices that need to be handled, i.e., m values instead of $|U|$ values
- There is a hitch: two keys may hash to the same slot
- Need a technique to resolve such collision conflicts
  - Various options: chaining, probing, double hashing

Hash tables/3

Collision resolution with chaining:

5 elements that hash to the same slot

Hash tables/3

Collision resolution with chaining:

- Chaining: put items that hash to same slot in linked list

Hash tables/3

Collision resolution with chaining:

- Chaining: put items that hash to same slot in linked list
- Each slot contains either `NIL` or a pointer to head of a list

Hash tables/3

Collision resolution with chaining:

- Chaining: put items that hash to same slot in linked list
- Each slot contains either `NIL` or a pointer to head of a list
- Search, insert, and delete: relatively easy to implement

EXERCISE SL07-1

Define a hash table with chaining for collision resolution in C.

a) bad

```
struct node {
    char name[20];
    struct node* next;
}

struct node HT[7000];
```

b) good

```
struct node {
    char name[20];
    struct node* next;
}

struct node* HT[7000];
```

Hash tables/4

Analysis of hashing

- Given a hash table HT with m slots that stores n items, define the load factor α of HT as n/m
  - α corresponds to average number of items in a list

Hash tables/4

Analysis of hashing

- Given a hash table HT with m slots that stores n items, define the load factor α of HT as n/m
  - α corresponds to average number of items in a list
- Assumption 1: any given item is equally likely to hash into any of the m slots of the hash table
  - This assumption is called *simple uniform hashing*

Hash tables/4

Analysis of hashing

- Given a hash table HT with m slots that stores n items, define the load factor α of HT as n/m
  - α corresponds to average number of items in a list
- Assumption 1: any given item is equally likely to hash into any of the m slots of the hash table
  - This assumption is called *simple uniform hashing*
- Assumption 2: $h(k)$ can be computed in constant time

HASH TABLES/4

implications: make HT big enough
(2–10x larger than needed)

Analysis of hashing

- Given a hash table HT with m slots that stores n items, define the load factor α of HT as n/m
  - α corresponds to average number of items in a list
- Assumption 1: any given item is equally likely to hash into any of the m slots of the hash table
  - This assumption is called *simple uniform hashing*
- Assumption 2: $h(k)$ can be computed in constant time
- Search (unsuccessful/successful) takes $\Theta(1+\alpha)$ time
  - Need probability theory to prove these two results

- best case: $n=1$ $m=1M$ $\Rightarrow \alpha = 1/1M \approx 0$, $\Theta(1)$
- worst case: $n=1M$ $m=1$ $\Rightarrow \alpha = 1M/1 = n$, $\Theta(n)$

EXERCISE SL07-2

What benefits do you expect by using a binary search tree instead of a linked list to handle hash table overflows?

(assumption: elements have order, can be compared)

- BST only pays off if there is lots of overflow
- hashing is not good / effective if there is lots of overflow

  reasons:
  - too small HT
  - bad hash function ($h(k)=0$)

- fix hashing instead

Table of contents — SL07

1. Hash tables

2. Hash functions
   - Goals and design
   - Division method
   - Multiplication method

3. Collision resolution

Hash functions/1

Design of hash functions

- Hash function h must be computed in constant time

Hash functions/1

Design of hash functions

- Hash function h must be computed in constant time
- Must satisfy the assumption of simple uniform hashing
  - Each key is equally likely to hash to any of the m slots

Hash functions/1

Design of hash functions

- Hash function h must be computed in constant time
- Must satisfy the assumption of simple uniform hashing
  - Each key is equally likely to hash to any of the m slots

Hashing non-integer keys

- Consider set $\mathbb{N}$ of natural numbers as universe of keys

Hash functions/1

Design of hash functions

- Hash function h must be computed in constant time
- Must satisfy the assumption of simple uniform hashing
  - Each key is equally likely to hash to any of the m slots

Hashing non-integer keys

- Consider set $\mathbb{N}$ of natural numbers as universe of keys
- If the keys are not natural number then a way is found to interpret them as such
  - Example: encoding of strings via ASCII character set

EXERCISE SL07-3/1

Give different options for mapping a string to an integer for the purpose of hashing. Discuss advantages and disadvantages.

ascii("RED") = 82 69 68

1) add ascii 82 + 69 + 68 = "RED" (result: "RED" = "ERD" = "DER")

2) concatenate ascii codes: 826968
radix notation (for 7 Bits)

$$(82)_{10\to 2} \cdot 128^2 + (69)_{10\to 2} \cdot 128^1 + (68)_{10\to 2} \cdot 128^0$$

$(82)_{10\to 2} = (1010010)_2$, $128^2 = 2^7$

3) ascii of the 1st character (result: "RED" = "R17" = "RIGHT")

EXERCISE SL07-3/2

Hash functions/2

Division method

- Hash function takes the remainder of k divided by m
  - $h(k) = (k \bmod m) + 1$
  - $m = 12, k = 100 \Rightarrow h(k) = 5$

Hash functions/2

Division method

- Hash function takes the remainder of k divided by m
  - $h(k) = (k \bmod m) + 1$
  - $m = 12, k = 100 \Rightarrow h(k) = 5$
- Generally, one tries to avoid certain choices for m
  - For $m = 2^p$, $h(k)$ gives the p least significant bits of k and all keys with the same ending go to the same slot

Hash functions/2

Division method

- Hash function takes the remainder of k divided by m
  - $h(k) = (k \bmod m) + 1$
  - $m = 12, k = 100 \Rightarrow h(k) = 5$
- Generally, one tries to avoid certain choices for m
  - For $m = 2^p$, $h(k)$ gives the p least significant bits of k and all keys with the same ending go to the same slot
- Good choice for m: prime number not close to 2^p
  - Hash table HT for roughly $n = 2000$ character strings
  - One may choose $m = 701$ because it is a prime near $\frac{2000}{3}$ (≈ 3 items per list) but not near any power of 2

EXERCISE SL07-4/1

Let $h(k) = k \mod 2^p - 1$ where k is a string interpreted in radix 2^p. Prove that strings a and b hash to the same value if a is a permutation of b.

EXERCISE SL07-4/2

EXERCISE SL07-4/3

Hash functions/3

Multiplication method

- Multiply key k by A and extract fractional part of kA
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor + 1$
  - A constant such that $0 < A < 1$
  - $kA \bmod 1$: fractional part of kA, i.e. $kA - \lfloor kA \rfloor$

Hash functions/3

Multiplication method

- Multiply key k by A and extract fractional part of kA
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor + 1$
  - A constant such that $0 < A < 1$
  - $kA \bmod 1$: fractional part of kA, i.e. $kA - \lfloor kA \rfloor$
- Advantage: value of m is not critical, typically use 2^p

Hash functions/3

Multiplication method

- Multiply key k by A and extract fractional part of kA
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor + 1$
  - A constant such that $0 < A < 1$
  - $kA \bmod 1$: fractional part of kA, i.e. $kA - \lfloor kA \rfloor$
- Advantage: value of m is not critical, typically use 2^p
- Optimal choice for A depends on data characteristics
  - Knuth suggests that $A \approx \frac{\sqrt{5}-1}{2} \approx 0.618$ works well

Hash functions/3

Multiplication method

- Multiply key k by A and extract fractional part of kA
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor + 1$
  - A constant such that $0 < A < 1$
  - $kA \bmod 1$: fractional part of kA, i.e. $kA - \lfloor kA \rfloor$
- Advantage: value of m is not critical, typically use 2^p
- Optimal choice for A depends on data characteristics
  - Knuth suggests that $A \approx \frac{\sqrt{5}-1}{2} \approx 0.618$ works well
- Example: assume that $A = 0.618$ and $m = 12$
  - $k = 102$: $h(k) = \lfloor 12 \cdot 0.036 \rfloor + 1 = \lfloor 0.432 \rfloor + 1 = 1$
  - $k = 103$: $h(k) = \lfloor 12 \cdot 0.654 \rfloor + 1 = \lfloor 7.848 \rfloor + 1 = 8$
  - $k = 104$: $h(k) = \lfloor 12 \cdot 0.272 \rfloor + 1 = \lfloor 3.264 \rfloor + 1 = 4$

Table of contents — SL07

1. Hash tables

2. Hash functions

3. Collision resolution
   Chaining
   Open addressing
   - Linear probing
   - Double hashing

Chaining

- The simplest approach is chaining.
- Chaining maintains a table of links, indexed by the keys, to lists of items with the same key.
- Pros: easy, flexible, hash table never gets full
- Cons: stores elements outside hash table, additional space for pointers

Open addressing

- The idea is to store all items in the hash table itself
  - Each table entry contains either an item or is empty
  - No lists and no items are stored outside the table

Open addressing

- The idea is to store all items in the hash table itself
  - Each table entry contains either an item or is empty
  - No lists and no items are stored outside the table
- A drawback is that the table can fill up: load factor α never exceeds 1

Open addressing

- The idea is to store all items in the hash table itself
  - Each table entry contains either an item or is empty
  - No lists and no items are stored outside the table
- A drawback is that the table can fill up: load factor α never exceeds 1
- Advantage of open addressing: avoid pointers
  - No need to follow pointers when accessing items
  - Simply compute sequence of slots to be examined
  - Larger number of slots for same amount of memory

Open addressing

- Open addressing uses an extended hash function that includes the probe number:

$$h\colon U \times \{1, 2, \ldots, m\} \to \{1, 2, \ldots, m\}$$

$\forall k$: $\langle h(k, 1), \ldots, h(k, m)\rangle$ is permutation of $\langle 1, \ldots, m\rangle$

Open addressing

- Open addressing uses an extended hash function that includes the probe number:

$$h\colon U \times \{1, 2, \ldots, m\} \to \{1, 2, \ldots, m\}$$

  $\forall k$: $\langle h(k, 1), \ldots, h(k, m)\rangle$ is permutation of $\langle 1, \ldots, m\rangle$
- All dictionary operations systematically probe table slots
- Sequence of positions probed depends on given key

Linear Probing/1

Linear probing

- Uses a hash function of the form
  - $h(k, i) = (h'(k) + ci) \bmod m$
  - where h' is a hash function

Linear Probing/1

Linear probing

- Uses a hash function of the form
  - $h(k, i) = (h'(k) + ci) \bmod m$
  - where h' is a hash function
- We try out (probe) a number of positions:
  NextLoc = (PreviousLoc + StepSize) mod m
- The step size should be chosen such that all locations are probed.
- Often a step size of one (c=1) is chosen. This guarantees that all locations are probed.

Linear Probing/2

- If the current location is used, try the next table location:

```
Algo: LinearProbingInsert(k)
---
if table is full then return error;
probe = h(k);
while HT[probe] is used do
  probe = (probe+1) mod m
table[probe] = k;
```

- Lookups walk along the table until the key or an empty slot is found
- Uses less memory than chaining: one does not have to store the links
- Slower than chaining: one might have to probe the table for a long time

EXERCISE SL07-5

Insert the keys 18, 41, 22, 44, 59, 32, 31, 73 into a hash table with 13 slots where linear probing with a step size of 1 is used as a collision resolution strategy. Show the final hash table.

EXERCISE SL07-6

Consider the following open addressing scheme:

```
1 i = h(k); j = 0;
2 if HT[i]==empty ∨ HT[i]==k then stop search;
3 j = (j+1) mod m; i = (i+j) mod m; goto 2;
```

Show that this schema is a quadratic probing scheme of the form $h(k,i) = (h(k) + c_1 i + c_2 i^2) \mod m$.

DOUBLE HASHING/1

- Two hash functions:
  1. h1 determines the initial location
  2. h2 determines the step size
  3. $h(k, i) = (h_1(k) + i * h_2(k) \bmod m$
- Initial probe is to position $HT[h_1(k)]$, successive probe positions are offset by $h_2(k)$.

DOUBLE HASHING/2

Search:

Algo: DoubleHashingInsert(HT,k)

```
i = 0;
repeat
    probe = (h1(k) + i*h2(k)) mod m;
    i = i+1;
until (HT[probe] is free ∨ i>m);
if i>m then Error: hash table overflow;
else HT[probe] = k;
```

DOUBLE HASHING/3

- $h_2(k)$ must be relatively prime to hash table size m
  - m a power of 2, h_2 always produces an odd number
  - m prime, h_2 always produces a positive integer $< m$

Double hashing/3

- $h_2(k)$ must be relatively prime to hash table size m
  - m a power of 2, h_2 always produces an odd number
  - m prime, h_2 always produces a positive integer $< m$
- Example: m prime and $h_2(k)$ positive integer $< m$
  - Let $h_1(k) = (k \bmod m) + 1$, $h_2(k) = (k \bmod m') + 1$
  - Choose m' to be slightly less than m (say $m' = m - 1$)
  - Given $k = 123456$, $m = 701$, $m' = 700$: $h_1(k) = 81$, $h_2(k) = 257$ (first probe to 81, then every 257th slot)

Double hashing/3

- $h_2(k)$ must be relatively prime to hash table size m
  - m a power of 2, h_2 always produces an odd number
  - m prime, h_2 always produces a positive integer $< m$
- Example: m prime and $h_2(k)$ positive integer $< m$
  - Let $h_1(k) = (k \bmod m) + 1$, $h_2(k) = (k \bmod m') + 1$
  - Choose m' to be slightly less than m (say $m' = m - 1$)
  - Given $k = 123456$, $m = 701$, $m' = 700$: $h_1(k) = 81$, $h_2(k) = 257$ (first probe to 81, then every 257th slot)
- Cost of searching in open-address hash table with $\alpha = n/m < 1$
  - unsuccessful: $1/(1-\alpha)$
  - successful: $\ln(1/(1-\alpha))/\alpha$

EXERCISE SL07-7

Insert the keys 18, 41, 22, 44, 59, 32, 31, 73 into a hash table with double hashing where

$$h1(k) = k \mod 13$$
$$h2(k) = 8 - (k \mod 8)$$
$$H(k,i) = (h1(k) + i * h2(k)) \mod 13$$

Double hashing/4

Dictionary operations

- Inserting and searching are relatively straightforward
  - At least if one assumes that there are no deletions

Double hashing/4

Dictionary operations

- Inserting and searching are relatively straightforward
  - At least if one assumes that there are no deletions
- Deletion is difficult: cannot mark deleted slot as empty
  - Solution: use special value to mark a deleted slot
  - This affects all hash table functions (e.g., the simple procedures for insertion must be extended)

EXERCISE SL07-8

Consider hashing with open addressing. Give an example that illustrates a possible problem when deleting an element rather than marking it as deleted. Can it make sense to attempt to delete elements instead of marking them as deleted? When is it possible to remove DEL?

Hashing with Open Addressing/1

Assume hashing with open addressing as a collision resolution strategy.

1. Define the hash table and give a procedure to initialize it.
2. Give a procedure to insert, search and delete an element

Hashing with Open Addressing/2

definition of hash table:

```
struct elem {
  int key;
  int status;
    /* 0: OCCupied, -1: EMPty, -2: DELeted */
};

struct elem HT [7000];
```

Algo: HTinit(HT)

for *i* = *0* **to** *m-1* **do** HT[i].status = EMP;

Hashing with Open Addressing/3

Algo: HTinsert(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨ HT[probe].status≠OCC;
if i≥m then return -1;
HT[probe].status = 0;
HT[probe].key = k;
return probe
```

Hashing with Open Addressing/4

Algo: HTsearch(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨
        HT[probe].status==OCC ∧ HT[probe].key==k ∨
        HT[probe].status==EMP;
if i≥m ∨ HT[probe].status==EMP then return -1;
return probe
```

Hashing with Open Addressing/5

Algo: HTdelete(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨
        HT[probe].status==EMP ∨
        HT[probe].status==OCC ∧ HT[probe].key==k;
if i≥m ∨ HT[probe].status==EMP then return -1;
HT[probe].status = DEL;
return probe
```

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$
- Probing, unsuccessful ($\alpha : 1/(1 - \alpha)$)
  - $0.2 : 1.25$, $0.5 : 2$, $0.8 : 5$, $0.9 : 10$

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$
- Probing, unsuccessful ($\alpha : 1/(1 - \alpha)$)
  - $0.2 : 1.25$, $0.5 : 2$, $0.8 : 5$, $0.9 : 10$
- Probing, successful ($\alpha : (\ln(1/(1 - \alpha))/\alpha$)
  - $0.2 : 0.28$, $0.5 : 1.39$, $0.8 : 2.01$, $0.9 : 2.56$

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$
- Probing, unsuccessful ($\alpha : 1/(1 - \alpha)$)
  - $0.2 : 1.25$, $0.5 : 2$, $0.8 : 5$, $0.9 : 10$
- Probing, successful ($\alpha : (\ln(1/(1 - \alpha))/\alpha$)
  - $0.2 : 0.28$, $0.5 : 1.39$, $0.8 : 2.01$, $0.9 : 2.56$

Final remarks

- Hashing is very efficient (not obvious, probability)

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$
- Probing, unsuccessful ($\alpha : 1/(1 - \alpha)$)
  - $0.2 : 1.25$, $0.5 : 2$, $0.8 : 5$, $0.9 : 10$
- Probing, successful ($\alpha : (\ln(1/(1 - \alpha))/\alpha)$
  - $0.2 : 0.28$, $0.5 : 1.39$, $0.8 : 2.01$, $0.9 : 2.56$

Final remarks

- Hashing is very efficient (not obvious, probability)
- May not be easy to determine parameters (e.g. m).
- Worst case exists and cannot be avoided in general.

EXERCISE SL07-9/1

Consider quadratic probing of the form

$$h(k,i) = (h(k) + i/2 + i^2/2) \mod 2^p$$

Prove that with this scheme every table position is examined in the worst case.

EXERCISE SL07-9/2

EXERCISE SL07-9/3

SUMMARY/1

- Hashing is simple to implement.
- Hashing requires some difficult to choose parameters (e.g., size of hash table).
- Hash table give constant access time (under some assumptions).
- Hash table reorganization (enlarging or shrinking) are expensive (not constant anymore).
- Analysis of hashing requires probability theory.

- In general, hashing must deal with collisions.
- In general, no special values are available to identify empty slots or deleted elements.

SUMMARY/2

- Hashing, collision resolutions:
  - $h(k,i) = (h_1(k) + c * i) \mod m$
  - $h(k,i) = (h_1(k) + c_1 * i + c_2 * i^2) \mod m$
  - $h(k,i) = (h_1(k) + h_2(k) * i) \mod m$
- Heuristic: spread data; avoid clustering of elements; get as few collisions as possible for all step sizes
- Example sequences of probed elements:
  - $4; 5; 7; -2; EMP$ (stops at EMP)
  - $4; DEL; 7; -2; EMP$ (DEL to avoid early stopping)
  - $-4; DEL; DEL; -2; EMP$
  - many DEL make hash table inefficient
  - probe sequence should visit all slots with m probes

Informatics II

Dynamic Programming
— SL08 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

Table of contents — SL08

1. Design techniques
   Design techniques
   Fibonacci numbers

2. Matrix multiplication

3. Dynamic programming

4. Longest common subsequence, coin changing

Design techniques/1

Design of algorithms

- Design techniques used so far
  - Iterative (brute-force) algorithms
    - Linear search, selection sort, insertion sort, etc
  - Data structure based algorithms
    - Heap sort, PQ sort, BST sort, RBT sort, etc
  - Divide-and-conquer algorithms
    - Binary search, merge sort, quick sort, etc

Design techniques/1

Design of algorithms

- Design techniques used so far
  - Iterative (brute-force) algorithms
    - Linear search, selection sort, insertion sort, etc
  - Data structure based algorithms
    - Heap sort, PQ sort, BST sort, RBT sort, etc
  - Divide-and-conquer algorithms
    - Binary search, merge sort, quick sort, etc
- But there exist other techniques
  - Dynamic programming (remember solutions)
    - Matrix multiplication, longest common subsequence
  - Greedy algorithms (choose local optimum in each step)
    - Graph based: minimum spanning tree, shortest path

Design techniques/2

Divide-and-conquer

- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. Divide: problem into a number of subproblems
  2. Conquer: subproblems by solving them recursively
  3. Combine: solutions of the subproblem into solution for the original problem

Design techniques/2

Divide-and-conquer

- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. Divide: problem into a number of subproblems
  2. Conquer: subproblems by solving them recursively
  3. Combine: solutions of the subproblem into solution for the original problem
- Subproblems are independent and non-overlapping

Design techniques/2

Divide-and-conquer

- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. Divide: problem into a number of subproblems
  2. Conquer: subproblems by solving them recursively
  3. Combine: solutions of the subproblem into solution for the original problem
- Subproblems are independent and non-overlapping

Dynamic programming

- Solve problem by combining solutions of subproblems

Design techniques/2

Divide-and-conquer

- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. Divide: problem into a number of subproblems
  2. Conquer: subproblems by solving them recursively
  3. Combine: solutions of the subproblem into solution for the original problem
- Subproblems are independent and non-overlapping

Dynamic programming

- Solve problem by combining solutions of subproblems
- Subproblems are not independent and overlapping

FIBONACCI NUMBERS/1

Recursive definition

$$\text{Fibonacci} \quad \mathit{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathit{fib}(n-2) & \\ + \mathit{fib}(n-1) & \text{if } n > 1 \end{cases}$$

FIBONACCI NUMBERS/1

Recursive definition

$$\text{Fibonacci} \quad \mathit{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathit{fib}(n-2) & \\ + \mathit{fib}(n-1) & \text{if } n > 1 \end{cases}$$

Recursive algorithm

Algo: Fib1(n)

---

if $n<2$ **then return** n;
return Fib1(n-2) + Fib1(n-1);

FIBONACCI NUMBERS/2

Analysis of algorithm

- Keep calculating the same values over and over again

EXERCISE SL08-1

Determine the asymptotic runtime complexity of the recursive Fibonacci numbers algorithm.

FIBONACCI NUMBERS/3

Bottom-up approach

- Keep **remembering solutions** of solved subproblems

Algo: Fib2(n)

```
f[1] = 0;
f[2] = 1;
for i = 3 to n+1 do
  f[i] = f[i-2] + f[i-1]
return f[n+1];
```

FIBONACCI NUMBERS/3

Bottom-up approach

- Keep **remembering solutions** of solved subproblems

Algo: Fib2(n)

```
f[1] = 0;
f[2] = 1;
for i = 3 to n+1 do
  f[i] = f[i-2] + f[i-1]
return f[n+1];
```

- Can compute value of $fib(n)$ in linear time (i.e., $\Theta(n)$)
- Algorithm computes the solution in bottom-up fashion.
- The bottom-up approach trades space for time.

FIBONACCI NUMBERS/4

Optimized bottom-up algorithm

- Can do without remembering all the previous results

Algo: Fib3(n)

```
s = 0;
t = 1;
for i = 1 to n do
    r = t;
    t = t+s;
    s = r
return s;
```

FIBONACCI NUMBERS/4

Optimized bottom-up algorithm

- Can do without remembering all the previous results

Algo: Fib3(n)

```
s = 0;
t = 1;
for i = 1 to n do
    r = t;
    t = t+s;
    s = r
return s;
```

- Thus, space complexity is constant rather than linear

Observations

- An inefficient algorithm does not imply that no efficient solution for the problem at hand exists.

Observations

- An inefficient algorithm does not imply that no efficient solution for the problem at hand exists.
- If the same problem is computed over and over again we can improve the performance by remembering previously computed solutions.

Observations

- An inefficient algorithm does not imply that no efficient solution for the problem at hand exists.
- If the same problem is computed over and over again we can improve the performance by remembering previously computed solutions.
- Such an approach trades space for time.

Observations

- An inefficient algorithm does not imply that no efficient solution for the problem at hand exists.
- If the same problem is computed over and over again we can improve the performance by remembering previously computed solutions.
- Such an approach trades space for time.
- The sequence of computation must be organized in such a way that later computed solutions can use earlier computed solutions.

TABLE OF CONTENTS — SL08

1. Design techniques

2. Matrix multiplication
   Basics of dynamic programming
   Matrix-chain multiplication

3. Dynamic programming

4. Longest common subsequence, coin changing

Overview of dynamic programming

- Invented in the 1950s by Richard Bellman as a general method for optimizing multistage decision procedures

Overview of dynamic programming

- Invented in the 1950s by Richard Bellman as a general method for optimizing multistage decision procedures
- The term *dynamic programming* (DP) refers to a tabular method

Overview of dynamic programming

- Invented in the 1950s by Richard Bellman as a general method for optimizing multistage decision procedures
- The term *dynamic programming* (DP) refers to a tabular method
- DP is typically applied to optimization problems
  - Have to choose one solution out of many
  - Want the solution with the optimal value
  - Characterize the structure of a solution
    - Consists of a sequence of choices that can be made
    - What choices have to be made for an optimal solution?
  - Algorithm computes an optimal value (plus, if needed, an optimal solution)

Matrix-chain multiplication/1

Problem definition

- Matrices $A(n \times m)$ and $B(m \times k)$ can be multiplied to get matrix $C(n \times k)$ using nmk scalar multiplications

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{ij} = \sum_{l=1}^{m} a_{il} \cdot b_{lj} \quad (\text{e.g. } c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22})$$

Matrix-chain multiplication/1

Problem definition

- Matrices $A(n \times m)$ and $B(m \times k)$ can be multiplied to get matrix $C(n \times k)$ using nmk scalar multiplications

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{ij} = \sum_{l=1}^{m} a_{il} \cdot b_{lj} \quad (\text{e.g. } c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22})$$

- Matrix multiplication is associative, i.e., $A(BC) = (AB)C$

Matrix-chain multiplication/1

Problem definition

- Matrices $A(n \times m)$ and $B(m \times k)$ can be multiplied to get matrix $C(n \times k)$ using nmk scalar multiplications

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{ij} = \sum_{l=1}^{m} a_{il} \cdot b_{lj} \quad (\text{e.g. } c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22})$$

- Matrix multiplication is associative, i.e., $A(BC) = (AB)C$
- Problem: compute product of many matrices efficiently (i.e., determine the best sequence of multiplications)

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1 A_2)(A_3 A_4)$ has cost $1200 + 10000 + 30000 = 41200$

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12}A_{34}$ is a (30×25) matrix

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12}A_{34}$ is a (30×25) matrix
      - computation cost of A_{1234} is $30 * 40 * 25 = 30'000$

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12}A_{34}$ is a (30×25) matrix
      - computation cost of A_{1234} is $30 * 40 * 25 = 30'000$
    - $((A_1A_2)A_3)A_4$ has cost $1200 + 12000 + 7500 = 20700$

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12}A_{34}$ is a (30×25) matrix
      - computation cost of A_{1234} is $30 * 40 * 25 = 30'000$
    - $((A_1A_2)A_3)A_4$ has cost $1200 + 12000 + 7500 = 20700$
    - $A_1((A_2A_3)A_4)$ has cost $400 + 250 + 750 = 1400$

Matrix-chain multiplication/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12}A_{34}$ is a (30×25) matrix
      - computation cost of A_{1234} is $30 * 40 * 25 = 30'000$
    - $((A_1A_2)A_3)A_4$ has cost $1200 + 12000 + 7500 = 20700$
    - $A_1((A_2A_3)A_4)$ has cost $400 + 250 + 750 = 1400$
- In other words, it is necessary to optimally parenthesize $A_1 \cdots A_n$.

EXERCISE SL08-2

Propose an appropriate problem representation for solving the matrix multiplication problem.

Matrix-chain multiplication/3

Optimal substructure

- Let $M(i,j)$ $(i \leq j)$ be the minimum number of scalar multiplications necessary to compute $A_{i\ldots j} = A_i \cdots A_j$
- Key observations
  - Outermost parenthesis partitions the chain of matrices (i,j) at some k $(i \leq k < j)$: $(A_i \cdots A_k)(A_{k+1} \cdots A_j)$
  - Optimal parenthesization of matrices (i,j) has optimal parenthesization on either side of k, i.e. for matrices (i,k) and $(k+1,j)$
- Example $(i = 1, j = n, k = 3)$:

$$\overbrace{\underbrace{(A_1 \times A_2 \times A_3)}_{\text{opt paranthesis here}} \times \underbrace{(A_4 \times \ldots \times A_n)}_{\text{opt parantheses here}}}^{\text{let this be opt paranthesis}}$$

Matrix-chain multiplication/4

Recursive problem formulation

- One can try out all possible k and take the one with the smallest cost:

$$M(i,j) = \min_{i \leq k < j} \{M(i,k) + M(k+1,j) + d_{i-1} d_k d_j\}$$

Matrix-chain multiplication/4

Recursive problem formulation

- One can try out all possible k and take the one with the smallest cost:

$$M(i,j) = \min_{i \leq k < j}\{M(i,k) + M(k+1,j) + d_{i-1}d_kd_j\}$$

- The base case is $M(i,i) = 0$
- A direct recursive implementation is exponential (there is a lot of duplicated work)

Matrix-chain multiplication/4

Recursive problem formulation

- One can try out all possible k and take the one with the smallest cost:

$$M(i,j) = \min_{i \leq k < j}\{M(i,k) + M(k+1,j) + d_{i-1}d_kd_j\}$$

- The base case is $M(i,i) = 0$
- A direct recursive implementation is exponential (there is a lot of duplicated work)
- But there are only few different subproblems (i,j) (one solution for each i and j, $i \leq j$)

Matrix-chain multiplication/5

Recursive solution

- Direct implementation of the recursive problem formulation

Algo: MatrixRec(i,j)

---

if *i==j* **then return** 0;
min = MatrixRec(i,i) + MatrixRec(i+1,j) + $d_{i-1}d_id_j$;
for *k = i+1* **to** *j-1* **do**
  q = MatrixRec(i,k) + MatrixRec(k+1,j) + $d_{i-1}d_kd_j$;
  if *q<min* **then** min = q;
return min;

Matrix-chain multiplication/6

DP solution

- Idea for dynamic programming solution: store the optimal cost $M(i, j)$ for each subproblem in a two-dimensional array $M[1 \ldots n, 1 \ldots n]$
  - Trivially, $M(i, i) = 0$ for $1 \leq i \leq n$
  - To compute $M(i, j)$ one needs only values of M for subproblems of length less than $l = j - i$
  - So, one has to solve subproblems according to their length: first subproblems of length 1, then the ones of length 2, and so on

Matrix-chain multiplication/6

DP solution

- Idea for dynamic programming solution: store the optimal cost $M(i, j)$ for each subproblem in a two-dimensional array $M[1 \ldots n, 1 \ldots n]$
  - Trivially, $M(i, i) = 0$ for $1 \leq i \leq n$
  - To compute $M(i, j)$ one needs only values of M for subproblems of length less than $l = j - i$
  - So, one has to solve subproblems according to their length: first subproblems of length 1, then the ones of length 2, and so on
- To reconstruct an optimal solution: for every (i, j), one records in $c[i, j]$ the optimal split into (i, k) and $(k + 1, j)$

EXERCISE SL08-3

Illustrate the data structure that is used to compute the optimal parentheses for a series of matrix multiplications. Illustrate the sequence in which the computation steps are performed.

Matrix-chain multiplication/7

A simple example

- Run the algorithm on the chain $\langle A_1, A_2, A_3, A_4 \rangle$
  - $A_1(3 \times 2)$, $A_2(2 \times 4)$, $A_3(4 \times 5)$, $A_4(5 \times 1)$
-
  - $M[1,2] = 3 \cdot 2 \cdot 4 = 24$
  - $M[2,3] = 2 \cdot 4 \cdot 5 = 40$
  - $M[3,4] = 4 \cdot 5 \cdot 1 = 20$

Matrix-chain multiplication/7

A simple example

- Run the algorithm on the chain $\langle A_1, A_2, A_3, A_4 \rangle$
  - $A_1(3 \times 2)$, $A_2(2 \times 4)$, $A_3(4 \times 5)$, $A_4(5 \times 1)$
-
  - $M[1,2] = 3 \cdot 2 \cdot 4 = 24$
  - $M[2,3] = 2 \cdot 4 \cdot 5 = 40$
  - $M[3,4] = 4 \cdot 5 \cdot 1 = 20$
-
  - $M[1,3] = M[1,2] + M[3,3] + 3 \cdot 4 \cdot 5 = 84$
  - $M[1,3] = M[1,1] + M[2,3] + 3 \cdot 2 \cdot 5 = 70$

Matrix-chain multiplication/7

A simple example

- Run the algorithm on the chain $\langle A_1, A_2, A_3, A_4 \rangle$
  - $A_1(3 \times 2)$, $A_2(2 \times 4)$, $A_3(4 \times 5)$, $A_4(5 \times 1)$
-
  - $M[1,2] = 3 \cdot 2 \cdot 4 = 24$
  - $M[2,3] = 2 \cdot 4 \cdot 5 = 40$
  - $M[3,4] = 4 \cdot 5 \cdot 1 = 20$
-
  - $M[1,3] = M[1,2] + M[3,3] + 3 \cdot 4 \cdot 5 = 84$
  - $M[1,3] = M[1,1] + M[2,3] + 3 \cdot 2 \cdot 5 = 70$
-
  - $M[2,4] = M[2,3] + M[4,4] + 2 \cdot 5 \cdot 1 = 50$
  - $M[2,4] = M[2,2] + M[3,4] + 2 \cdot 4 \cdot 1 = 28$

Matrix-chain multiplication/7

A simple example

- Run the algorithm on the chain $\langle A_1, A_2, A_3, A_4 \rangle$
  - $A_1(3 \times 2)$, $A_2(2 \times 4)$, $A_3(4 \times 5)$, $A_4(5 \times 1)$

-
  - $M[1,2] = 3 \cdot 2 \cdot 4 = 24$
  - $M[2,3] = 2 \cdot 4 \cdot 5 = 40$
  - $M[3,4] = 4 \cdot 5 \cdot 1 = 20$
-
  - $M[1,3] = M[1,2] + M[3,3] + 3 \cdot 4 \cdot 5 = 84$
  - $M[1,3] = M[1,1] + M[2,3] + 3 \cdot 2 \cdot 5 = 70$
-
  - $M[2,4] = M[2,3] + M[4,4] + 2 \cdot 5 \cdot 1 = 50$
  - $M[2,4] = M[2,2] + M[3,4] + 2 \cdot 4 \cdot 1 = 28$
-
  - $M[1,4] = M[1,3] + M[4,4] + 3 \cdot 5 \cdot 1 = 85$
  - $M[1,4] = M[1,2] + M[3,4] + 3 \cdot 4 \cdot 1 = 56$
  - $M[1,4] = M[1,1] + M[2,4] + 3 \cdot 2 \cdot 1 = 34$

EXERCISE SL08-4

Determine the optimal parentheses for computing the matrix product $A_1(3 \times 2) \times A_2(2 \times 4) \times A_3(4 \times 5) \times A_4(5 \times 1)$.

	1	2	3	4
1				
2				
3				
4				

	1	2	3	4
1				
2				
3				
4				

	1	2	3	4
1				
2				
3				
4				

	1	2	3	4
1				
2				
3				
4				

EXERCISE SL08-5

$M[i, j]$ is used to determine the **cost** of the optimal parentheses. Where to actually put the parentheses is not stored. We need an additional data structure for this that is similar to M. Extract this information from the previously computed M and fill the following matrix c.

	1	2	3	4
1				
2				
3				
4				

Matrix-chain multiplication/8

DP solution

Algo: MatrixDP(d)

```
for i = 1 to n do M[i,i] = 0;
for l = 1 to n-1 do
    for i = 1 to n-l do
        j = i+l;
        M[i,j] = ∞;
        for k = i to j-1 do
            q = M[i,k] + M[k+1,j] + d_{i-1} d_k d_j;
            if q<M[i,j] then M[i,j] = q; c[i,j] = k;
return (M,c);
```

MATRIX-CHAIN MULTIPLICATION/9

Analysis of the DP algorithm

- After the execution of the algorithm it holds that
  - $M[1, n]$ contains the value of the optimal solution
  - c stores the optimal subdivisions (choices of k)

Matrix-chain multiplication/9

Analysis of the DP algorithm

- After the execution of the algorithm it holds that
  - $M[1, n]$ contains the value of the optimal solution
  - c stores the optimal subdivisions (choices of k)
- It is easy to see that the running time is $O(n^3)$
  - Note that the algorithm has three nested loops
  - Reduction from exponential to polynomial time

Matrix-chain multiplication/10

Memoization/1

- Memoization is a technique for DP problems that preserves the recursive structure but does not do redundant computation.
- During the recursive calls newly computed solutions are stored in the DP data structure.
- The recursive procedure first checks if the solution has already been computed. If so it return the stored result.
- An advantage is that the exact computation sequence must not be programmed explicitly.

MATRIX-CHAIN MULTIPLICATION/11

Memoization/2

- Can structure solution also as a recursive algorithm

Algo: MatrixMemo(d,i,j)

---

if i==j **then return** 0;
if $M[i,j] < \infty$ **then return** M[i,j];
for k = i **to** j-1 **do**
  q = $d_{i-1} d_k d_j$ + MatrixMemo(d,i,k) + MatrixMemo(d,k+1,j);
  if q<$M[i,j]$ **then** M[i,j] = q;
return M;

MATRIX-CHAIN MULTIPLICATION/11

Memoization/2

- Can structure solution also as a recursive algorithm

Algo: MatrixMemo(d,i,j)

---

if i==j **then return** 0;
if $M[i,j] < \infty$ **then return** M[i,j];
for k = i **to** j-1 **do**
  q = $d_{i-1} d_k d_j$ + MatrixMemo(d,i,k) + MatrixMemo(d,k+1,j);
  if q<$M[i,j]$ **then** M[i,j] = q;
return M;

- Initialize all items to ∞ and run program ($i = 1$, $j = n$)

EXERCISE SL08-6/1

Use recursion and memoization to determine the number of moves that is required to solve the tower of Hanoi: moving all disks from peg 1 to peg 2 one disk at a time and never placing a larger disk on a smaller disk.

EXERCISE SL08-6/2

TABLE OF CONTENTS — SL08

1. Design techniques

2. Matrix multiplication

3. Dynamic programming
   Dynamic programming in detail

4. Longest common subsequence, coin changing

Dynamic programming/1

- We typically apply dynamic programming to optimization problems.
- The goal is to find a solution with the optimal (minimum or maximum) value.
- Dynamic programming is used frequently in bioinformatics (sequence alignment) and economics (macroeconomic models).
- Dynamic programming solves problems by combining the solutions to subproblems.
- Dynamic programming applies when subproblems overlap.
- A dynamic-programming algorithm solves each subproblem just once.

Dynamic programming/2

- **Step 1**: Show optimal substructure – an optimal solution to a problem contains optimal solutions to subproblems
  - Solution to a problem
    - Making a choice out of a number of possibilities (look what possible choices there can be)
    - Solving one or more subproblems that are the result of such a choice (characterize the space of subproblems)
  - Show that solutions to subproblems must themselves be optimal for the whole solution to be optimal

DYNAMIC PROGRAMMING/2

- **Step 1**: Show optimal substructure – an optimal solution to a problem contains optimal solutions to subproblems
  - Solution to a problem
    - Making a choice out of a number of possibilities (look what possible choices there can be)
    - Solving one or more subproblems that are the result of such a choice (characterize the space of subproblems)
  - Show that solutions to subproblems must themselves be optimal for the whole solution to be optimal

- **Step 2**: Give a recursive problem formulation for getting the optimal value
  - $M_{opt} = \min_k\{$(combination of M_{opt} of all subproblems resulting from the choice k) + (cost associated with making the choice k)$\}$

Dynamic programming/3

- **Step 3**: Compute the value of an optimal solution in a bottom-up fashion (or use memoization)

Dynamic programming/3

- **Step 3**: Compute the value of an optimal solution in a bottom-up fashion (or use memoization)
- Show that the number of different instances of subproblems is bounded by a polynomial
- Check whether it is possible to reduce the space requirements (forget no longer needed solutions)

Dynamic programming/3

- **Step 3**: Compute the value of an optimal solution in a bottom-up fashion (or use memoization)
- Show that the number of different instances of subproblems is bounded by a polynomial
- Check whether it is possible to reduce the space requirements (forget no longer needed solutions)

- **Step 4**: If required construct an optimal solution from the computed information (sequence of choices made)

Dynamic programming/3

- **Step 3**: Compute the value of an optimal solution in a bottom-up fashion (or use memoization)
- Show that the number of different instances of subproblems is bounded by a polynomial
- Check whether it is possible to reduce the space requirements (forget no longer needed solutions)

- **Step 4**: If required construct an optimal solution from the computed information (sequence of choices made)

Table of contents — SL08

1. Design techniques

2. Matrix multiplication

3. Dynamic programming

4. Longest common subsequence, coin changing
   Longest common subsequence (LCS)
   Coin changes

Longest common subsequence/1

Problem definition

- Two text strings are given: X and Y

Longest common subsequence/1

Problem definition

- Two text strings are given: X and Y
- Want to quantify how similar they are
  - Spell checkers, DNA sequences

Longest common subsequence/1

Problem definition

- Two text strings are given: X and Y
- Want to quantify how similar they are
  - Spell checkers, DNA sequences
- One of the measures of similarity is the length of a longest common subsequence (LCS)

Longest common subsequence/1

Problem definition

- Two text strings are given: X and Y
- Want to quantify how similar they are
  - Spell checkers, DNA sequences
- One of the measures of similarity is the length of a longest common subsequence (LCS)
- Z is a subsequence of X if it is possible to generate Z by skipping some (possibly none) characters from X
  - LCS ("ACGGTTA", "CGTAT") = "CGTA" or "CGTT"

Longest common subsequence/1

Problem definition

- Two text strings are given: X and Y
- Want to quantify how similar they are
  - Spell checkers, DNA sequences
- One of the measures of similarity is the length of a longest common subsequence (LCS)
- Z is a subsequence of X if it is possible to generate Z by skipping some (possibly none) characters from X
  - LCS ("ACGGTTA", "CGTAT") = "CGTA" or "CGTT"
- To solve the LCS problem one has to find
  - "skips" that generate LCS (X, Y) from X
  - "skips" that generate LCS (X, Y) from Y

Longest common subsequence/2

Optimal substructure

- Start with empty Z and proceed from the ends of $X_n = x_1 x_2 \ldots x_n$ and $Y_m = y_1 y_2 \ldots y_m$

Longest common subsequence/2

Optimal substructure

- Start with empty Z and proceed from the ends of $X_n = x_1 x_2 \ldots x_n$ and $Y_m = y_1 y_2 \ldots y_m$
- One needs to consider two cases:
  - If $x_n = y_m$, append this symbol to the beginning of Z and find an optimal solution for LCS (X_{n-1}, Y_{m-1})
  - If $x_n \neq y_m$, either skip a letter from X_n or skip a letter from Y_m and decide about the choice by comparing LCS (X_{n-1}, Y_m) and LCS (X_n, Y_{m-1})

Longest common subsequence/2

Optimal substructure

- Start with empty Z and proceed from the ends of $X_n = x_1 x_2 \ldots x_n$ and $Y_m = y_1 y_2 \ldots y_m$
- One needs to consider two cases:
  - If $x_n = y_m$, append this symbol to the beginning of Z and find an optimal solution for LCS (X_{n-1}, Y_{m-1})
  - If $x_n \neq y_m$, either skip a letter from X_n or skip a letter from Y_m and decide about the choice by comparing LCS (X_{n-1}, Y_m) and LCS (X_n, Y_{m-1})
- The algorithm can be extended to allow more editing operations (e.g. changing a letter)

Longest common subsequence/3

Recursive problem formulation

- Let $c[i,j] = \text{LCS}\,(X_i, Y_j)$ with $c[0\ldots n, 0\ldots m]$

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \wedge x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } i,j > 0 \wedge x_i \neq y_j \end{cases}$$

Longest common subsequence/3

Recursive problem formulation

- Let $c[i,j] = \text{LCS}\,(X_i, Y_j)$ with $c[0\ldots n, 0\ldots m]$

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \wedge x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } i,j > 0 \wedge x_i \neq y_j \end{cases}$$

- The conditions in the problem restrict the subproblems
  - If $x_i = y_j$, one considers the subproblem of finding the LCS of X_{i-1} and Y_{j-1}
  - If $x_i \neq y_j$, one considers the subproblems of finding the LCS of X_{i-1} and Y_j and of X_i and Y_{j-1}

EXERCISE SL08-7

Give a recursive solution for solving the LCS problem.

EXERCISE SL08-8

Give a memoization solution for solving the LCS problem.

EXERCISE SL08-9

Determine the length of LCS ("GGTTCAT", "GTATCT").

	-	G	T	A	T	C	T
-							
G							
G							
T							
T							
C							
A							
T							

Longest common subsequence/4

DP algorithm:

Algo: LCSdyn(X_n,Y_m)

```
for i = 1 to n do c[i,0] = 0;
for j = 0 to m do c[0,j] = 0;
for i = 1 to n do
    for j = 1 to m do
        if x_i == y_j then
            c[i,j] = c[i-1,j-1] + 1
        else
            if c[i-1,j]≥c[i,j-1] then
                c[i,j] = c[i-1,j]
            else
                c[i,j] = c[i,j-1]
return c;
```

EXERCISE SL08-10

Print the longest common subsequence of two given text strings.

Coin Changes/1

Problem illustration

- Assume you have coins of different denominations, e.g., 5, 15, 20, 50.
- Assume that of each denomination you have an unlimited number of coins.
- Given an amount A and m denominations the goal is to change A with as few coins as possible.
- Example:
  - Denominations: 1, 2, 5, 10
  - A = 27, change = 2*10 + 1*5 + 1*2

Coin Changes/2

Problem definition

Given are coins of different denominations (e.g., 1, 5, 15, 20, 50, etc). The goal is to pay out amount A with the smallest possible numbers of coins. Develop a solution.

Three cases:

1. No solution: e.g., A=3 and the only available coin is 2.
2. Greedy solution (i.e., in each step use the highest possible coin) gives the optimal solution. Typically, denominations are designed in such a way.
3. DP solution if the optimal solution cannot be found by choosing the largest possible coin in each step (e.g., A=12, coins = (10, 6, 1))

EXERCISE SL08-11/1

Optimal substructure:

Problem representation:

EXERCISE SL08-11/2

Recursive problem formulation:

Example: change of amount $A = 20$ with coins 10, 6, 1

EXERCISE SL08-11/3

Give a recursive solution for solving the coin changing problem.

EXERCISE SL08-11/4

Give a recursive solution with memoizationfor solving the coin changing problem.

EXERCISE SL08-11/5

Give a dynamic programming solution for solving the coin changing problem.

SUMMARY

- Dynamic programming (DP) is an important algorithmic design technique.
- DP is used to solve optimization problems.
- DP applies to problems that can be subdivided into optimal subproblems.

- First, come up with a recursive problem formulation.
- Organize computation in a bottom-up fashion.
- Precisely specify meanings of cells in the DP table.

Informatik II

Graph Algorithms
— SL09 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

Table of contents — SL09

1. Graphs
   - Definition and terminology
   - Graph representation
   - Graph search

2. Breadth-first search (BFS)

3. Depth-first search (DFS)

4. Topological sorting

Definition and terminology/1

- A **graph** $G = (V, E)$ consists of
  - a finite set of **vertices** V
  - a set of edges $E \subseteq V \times V$

Definition and terminology/1

- A **graph** $G = (V, E)$ consists of
  - a finite set of **vertices** V
  - a set of edges $E \subseteq V \times V$
- An **edge** $e = (u, v)$ is a pair of vertices

Definition and terminology/1

- A **graph** $G = (V, E)$ consists of
  - a finite set of **vertices** V
  - a set of edges $E \subseteq V \times V$
- An **edge** $e = (u, v)$ is a pair of vertices
- Generally assume **directed graphs** where edges are directed (go from one node to another node)
- **Undirected graphs**:
  - edges are unordered pairs $\{u, v\}$
  - $V = \{a, b, c, d\}$ & $E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}$
  - Undirected graphs can be implemented with edges in both directions (as part of data structure or algorithms)

Definition and terminology/2

- Vertex v is **adjacent** to vertex u iff $(u, v) \in E$

Definition and terminology/2

- Vertex v is **adjacent** to vertex u iff $(u, v) \in E$
- The **degree** of a vertex is defined as follows
  - number of adjacent vertices (undirected graph)
  - number of incoming and outgoing edges (directed graph)
    - **in-degree**: number of incoming edges
    - **out-degree**: number of outgoing edges

versus

Definition and terminology/2

- Vertex v is **adjacent** to vertex u iff $(u, v) \in E$
- The **degree** of a vertex is defined as follows
  - number of adjacent vertices (undirected graph)
  - number of incoming and outgoing edges (directed graph)
    - **in-degree**: number of incoming edges
    - **out-degree**: number of outgoing edges

versus

- A **path** of length k is a sequence $\langle v_0, \ldots, v_k \rangle$ of vertices such that, for all $i \in \{1, \ldots, k\}$, v_i is adjacent to v_{i-1}

Definition and terminology/3

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$
- **Cycle**: a simple path $\langle v_0, \ldots, v_k \rangle$, except that $v_k = v_0$
- **Subgraph**: subset of vertices and edges that forms a graph

Definition and terminology/3

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$
- **Cycle**: a simple path $\langle v_0, \ldots, v_k \rangle$, except that $v_k = v_0$
- **Subgraph**: subset of vertices and edges that forms a graph

- **Connected graph**: every pair of vertices is connected by a path (undirected edges)
- **Connected component**: connected subgraph of maximal size
- **Tree**: connected graph with n vertices and $n - 1$ edges (undirected edges)
- **Forest**: collection of trees

Graph representation/1

- Adjacency lists

Graph representation/1

- Adjacency lists

Graph representation/1

- Adjacency lists

- Adjacency list of v: sequence of vertices adjacent to it

Graph representation/1

- Adjacency lists

- Adjacency list of v: sequence of vertices adjacent to it
- Represent graph by the adjacency lists of its vertices

Graph representation/1

- Adjacency lists

- Adjacency list of v: sequence of vertices adjacent to it
- Represent graph by the adjacency lists of its vertices
- Space requirements: $\Theta(|V| + |E|)$

EXERCISE SL09-1

Give an adjacency list representation of a nearly complete binary tree with 4 nodes.

Graph representation/2

- Adjacency matrix

Graph representation/2

- Adjacency matrix

	a	b	c	d
a	0	0	1	0
b	1	0	0	0
c	0	1	0	1
d	0	0	0	0

Graph representation/2

- Adjacency matrix

	a	b	c	d
a	0	0	1	0
b	1	0	0	0
c	0	1	0	1
d	0	0	0	0

- Represent graph by an adjacency matrix $M(|V| \times |V|)$
  - $M[x, y] = 1$ if there is an edge $e = (x, y)$ in the graph
  - $M[x, y] = 0$ if there is no edge $e = (x, y)$ in the graph

Graph representation/2

- Adjacency matrix

	a	b	c	d
a	0	0	1	0
b	1	0	0	0
c	0	1	0	1
d	0	0	0	0

- Represent graph by an adjacency matrix $M(|V| \times |V|)$
  - $M[x, y] = 1$ if there is an edge $e = (x, y)$ in the graph
  - $M[x, y] = 0$ if there is no edge $e = (x, y)$ in the graph
- Space requirements: $\Theta(|V|^2)$

EXERCISE SL09-2

Consider an adjacency list representation of a directed graph. How long does it take to compute the in- and out-degree of a vertex?

Graph search/1

Graph search algorithms

- Systematic search of every edge and vertex of a graph

Graph search/1

Graph search algorithms

- Systematic search of every edge and vertex of a graph
- Applications: compilers, maze solving, network routing

Graph search/1

Graph search algorithms

- Systematic search of every edge and vertex of a graph
- Applications: compilers, maze solving, network routing

Pseudocode conventions

- Each node has certain properties (fields of a record)
  - adj: list of adjacent nodes
  - col: color of the node (B, G, W)
  - $pred$: predecessor node in traversal
  - $dist$: distance from the start node (BFS)
  - $stime$: time when a node is first visited (DFS)
  - $etime$: time when a node is last visited (DFS)
  - ... (more fields as required by application)

Graph search/2

Coloring is used to color vertices during a traversal. The colors allow to infer properties about the graph.

- Color changes during traversal

Graph search/2

Coloring is used to color vertices during a traversal. The colors allow to infer properties about the graph.

- Color changes during traversal
- Vertex v is white (i.e., $v.col = W$)
  - Means that v is undiscovered

GRAPH SEARCH/2

Coloring is used to color vertices during a traversal. The colors allow to infer properties about the graph.

- Color changes during traversal
- Vertex v is white (i.e., $v.col = W$)
  - Means that v is undiscovered
- Vertex v is gray (i.e., $v.col = G$)
  - Means that v has been discovered but not all adjacent vertices have been explored

Graph search/2

Coloring is used to color vertices during a traversal. The colors allow to infer properties about the graph.

- Color changes during traversal
- Vertex v is white (i.e., $v.col = W$)
  - Means that v is undiscovered
- Vertex v is gray (i.e., $v.col = G$)
  - Means that v has been discovered but not all adjacent vertices have been explored
- Vertex v is black (i.e., $v.col = B$)
  - Means that v has been discovered and all the adjacent vertices have been explored
  - In other words, the adjacency list of v was examined completely

Table of contents — SL09

1. Graphs

2. Breadth-first search (BFS)
   - BFS algorithm
   - BFS example
   - BFS properties

3. Depth-first search (DFS)

4. Topological sorting

BFS ALGORITHM/1

- A **breadth-first search** (BFS) starts at vertex s and visits all vertices reachable from s
  - In doing so BFS defines a **spanning tree** (a tree that includes all vertices).

BFS ALGORITHM/1

- A **breadth-first search** (BFS) starts at vertex s and visits all vertices reachable from s
  - In doing so BFS defines a **spanning tree** (a tree that includes all vertices).
- BFS is like walking in a labyrinth with a string and exploring the **entire** neighborhood first

BFS ALGORITHM/1

- A **breadth-first search** (BFS) starts at vertex s and visits all vertices reachable from s
  - In doing so BFS defines a **spanning tree** (a tree that includes all vertices).
- BFS is like walking in a labyrinth with a string and exploring the **entire** neighborhood first
- The starting vertex s is assigned distance 0

BFS ALGORITHM/1

- A **breadth-first search** (BFS) starts at vertex s and visits all vertices reachable from s
  - In doing so BFS defines a **spanning tree** (a tree that includes all vertices).
- BFS is like walking in a labyrinth with a string and exploring the **entire** neighborhood first
- The starting vertex s is assigned distance 0
- In the first round the string is unrolled 1 unit
  - All vertices that are one edge away from the starting vertex s are visited and assigned distance 1

BFS ALGORITHM/2

- In the second round the string is unrolled 2 units
  - All vertices that are two edges away from the starting vertex s are visited and assigned distance 2
  - Thus, all the vertices that can be reached by unrolling the string 2 units are assigned distance 2

BFS ALGORITHM/2

- In the second round the string is unrolled 2 units
  - All vertices that are two edges away from the starting vertex s are visited and assigned distance 2
  - Thus, all the vertices that can be reached by unrolling the string 2 units are assigned distance 2
- This continues until every reachable vertex has been assigned a distance

BFS ALGORITHM/2

- In the second round the string is unrolled 2 units
  - All vertices that are two edges away from the starting vertex s are visited and assigned distance 2
  - Thus, all the vertices that can be reached by unrolling the string 2 units are assigned distance 2
- This continues until every reachable vertex has been assigned a distance
- The label of any vertex v corresponds to the length of the shortest path from s to v

BFS ALGORITHM/3

Algo: BFS(G,s)

---

foreach $v \in G.V$ **do**
  v.col = W; v.dist = ∞; v.pred = NIL

s.col = G; s.dist = 0;
InitQueue(Q);
Enqueue(Q,s);
while $Q \neq \emptyset$ **do**
  v = Dequeue(Q);
  foreach $u \in v.adj$ **do**
    if *u.col*==*W* **then**
      u.col = G; u.dist = v.dist+1; u.pred = v; Enqueue(Q,u)
  v.col = B;

EXERCISE SL09-3

Show the distances and predecessors that result from a BFS starting at node 3.

Illustrate Q during the BFS. Determine the size of Q in the general case.

BFS EXAMPLE/1

BFS example/1

start)

BFS EXAMPLE/1

BFS example/1

start)

1)

BFS EXAMPLE/1

BFS example/1

BFS EXAMPLE/1

BFS example/1

BFS EXAMPLE/1

BFS example/1

BFS EXAMPLE/2

BFS example/2

5)

BFS EXAMPLE/2

BFS example/2

5)

6)

BFS EXAMPLE/2

BFS example/2

5)

6)

7)

BFS EXAMPLE/2

BFS example/2

BFS EXAMPLE/2

BFS example/2

EXERCISE SL09-4

Does the ordering of adjacency lists affect

1. the distances from the starting node computed by BFS?
2. the BFS tree?

BFS PROPERTIES/1

BFS running time

- Given a graph $G = (V, E)$
  - Initializing the vertices takes time $O(|V|)$
  - Vertices are enqueued if their color is white
    - Assuming that en- and dequeueing takes time $O(1)$, the total cost of this operation is $O(|V|)$
  - The adjacency list of a vertex is scanned when the vertex is dequeued
    - Since the sum of the lengths of all the adjacency lists is $O(|E|)$, the cost of scanning is $O(|E|)$

BFS PROPERTIES/1

BFS running time

- Given a graph $G = (V, E)$
  - Initializing the vertices takes time $O(|V|)$
  - Vertices are enqueued if their color is white
    - Assuming that en- and dequeueing takes time $O(1)$, the total cost of this operation is $O(|V|)$
  - The adjacency list of a vertex is scanned when the vertex is dequeued
    - Since the sum of the lengths of all the adjacency lists is $O(|E|)$, the cost of scanning is $O(|E|)$
- Total running time: $O(|V| + |E|)$ (linear in the size of the adjacency list representation of G)

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

- BFS discovers **all vertices reachable** from a source vertex s.

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

- BFS discovers **all vertices reachable** from a source vertex s.
- BFS computes the **shortest distance** to all reachable vertices.

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

- BFS discovers **all vertices reachable** from a source vertex s.
- BFS computes the **shortest distance** to all reachable vertices.
- BFS computes a **breadth-first tree** that contains every reachable vertex.

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

- BFS discovers **all vertices reachable** from a source vertex s.
- BFS computes the **shortest distance** to all reachable vertices.
- BFS computes a **breadth-first tree** that contains every reachable vertex.
- For any vertex v reachable from s, the path from s to v in the breadth-first tree corresponds to a **shortest path** in G.

Table of contents — SL09

1. Graphs

2. Breadth-first search (BFS)

3. Depth-first search (DFS)
   - DFS algorithm
   - DFS example
   - DFS properties

4. Topological sorting

DFS ALGORITHM/1

- A **depth-first search** (DFS) is like walking in a labyrinth with a string and following one path to the end:
  - Start at vertex v, tie the end of the string to v and paint v gray.
  - Set v as the current vertex and consider an arbitrary edge (v, u).
  - If the chosen edge leads to a non-white vertex u then return to v.
  - Otherwise, unroll the string, move to u, paint u gray, set u as current vertex, and repeat the previous steps.

DFS ALGORITHM/2

- Eventually, one will get to a point where all edges from u lead to non-white vertices.

DFS ALGORITHM/2

- Eventually, one will get to a point where all edges from u lead to non-white vertices.
- Then, one backtracks by rolling up the string until one gets back to a previously visited vertex v.

DFS ALGORITHM/2

- Eventually, one will get to a point where all edges from u lead to non-white vertices.
- Then, one backtracks by rolling up the string until one gets back to a previously visited vertex v.
- v thus becomes the new current vertex and one can repeat the previous steps.

DFS ALGORITHM/2

DFS annotations

- A DFS can be used to annotate vertices and edges with additional information:
  - start time $stime$ (when the vertex was visited first)
  - end time $etime$ (when the vertex was visited last)
  - edge classification (tree, forward, back, or cross edge)

DFS ALGORITHM/2

DFS annotations

- A DFS can be used to annotate vertices and edges with additional information:
  - start time $stime$ (when the vertex was visited first)
  - end time $etime$ (when the vertex was visited last)
  - edge classification (tree, forward, back, or cross edge)
- Such annotations reveal useful information about the graph that can be used by advanced algorithms.

DFS ALGORITHM/2

DFS annotations

- A DFS can be used to annotate vertices and edges with additional information:
  - start time $stime$ (when the vertex was visited first)
  - end time $etime$ (when the vertex was visited last)
  - edge classification (tree, forward, back, or cross edge)
- Such annotations reveal useful information about the graph that can be used by advanced algorithms.

DFS time stamping

- Vertex v is white before $v.stime$, gray between $v.stime$ and $v.etime$, and black after $v.etime$.

DFS ALGORITHM/2

DFS annotations

- A DFS can be used to annotate vertices and edges with additional information:
  - start time $stime$ (when the vertex was visited first)
  - end time $etime$ (when the vertex was visited last)
  - edge classification (tree, forward, back, or cross edge)
- Such annotations reveal useful information about the graph that can be used by advanced algorithms.

DFS time stamping

- Vertex v is white before $v.stime$, gray between $v.stime$ and $v.etime$, and black after $v.etime$.
- Gray vertices form a linear chain and the traversal of the nodes works like a stack (LIFO).

DFS ALGORITHM/3

Algo: DFS-Tree(G,s)

```
s.col = G; time = time+1; s.time = time;
foreach u ∈ s.adj do
    if u.col==W then
        u.pred = s;
        DFS-Tree(G,u)
s.col = B; time = time+1; s.etime = time;
```

DFS ALGORITHM/4

Algo: DFS(G)

---

foreach $v \in G.V$ **do**
  v.col = W; v.pred = NIL

time = 0;
foreach $v \in G.V$ **do**
  if *v.col*==*W* **then** DFS-Tree(G,v);

DFS ALGORITHM/4

Algo: DFS(G)

foreach $v \in G.V$ **do**
  v.col = W; v.pred = NIL

time = 0;
foreach $v \in G.V$ **do**
  if *v.col*==*W* **then** DFS-Tree(G,v);

Some observations

- DFS guarantees that every vertex is visited

DFS ALGORITHM/4

Algo: DFS(G)

foreach $v \in G.V$ **do**
  v.col = W; v.pred = NIL

time = 0;
foreach $v \in G.V$ **do**
  if *v.col*==*W* **then** DFS-Tree(G,v);

Some observations

- DFS guarantees that every vertex is visited
- Each call of `DFS-Tree` adds a new tree to depth-first forest

DFS ALGORITHM/4

Algo: DFS(G)

---

foreach $v \in G.V$ **do**
  v.col = W; v.pred = NIL

time = 0;
foreach $v \in G.V$ **do**
  if *v.col*==*W* **then** DFS-Tree(G,v);

Some observations

- DFS guarantees that every vertex is visited
- Each call of `DFS-Tree` adds a new tree to depth-first forest
- After termination, every vertex has a start and end time

DFS EXAMPLE/1

DFS search with start and end times:

start)

DFS EXAMPLE/1

DFS search with start and end times:

start)

1)

DFS EXAMPLE/1

DFS search with start and end times:

start)

1)

2)

DFS EXAMPLE/1

DFS search with start and end times:

DFS EXAMPLE/1

DFS search with start and end times:

DFS EXAMPLE/1

DFS search with start and end times:

DFS EXAMPLE/2

6)

DFS EXAMPLE/2

6)

7)

DFS EXAMPLE/2

6)

7)

8)

DFS EXAMPLE/2

DFS EXAMPLE/2

DFS EXAMPLE/2

DFS EXAMPLE/2

DFS EXAMPLE/2

DFS PROPERTIES/1

DFS running time

- Given a graph $G = (V, E)$
  - Each loop in `DFS` takes time $O(|V|)$, excluding the time to execute `DFS-Tree`
  - `DFS-Tree` is called once for every vertex
    - It is only invoked on white vertices
    - It paints the vertex gray immediately
  - Each time, `DFS-Tree` explores an adjacency list
  - Hence, the overall cost of `DFS-Tree` is $O(|E|)$

DFS PROPERTIES/1

DFS running time

- Given a graph $G = (V, E)$
  - Each loop in `DFS` takes time $O(|V|)$, excluding the time to execute `DFS-Tree`
  - `DFS-Tree` is called once for every vertex
    - It is only invoked on white vertices
    - It paints the vertex gray immediately
  - Each time, `DFS-Tree` explores an adjacency list
  - Hence, the overall cost of `DFS-Tree` is $O(|E|)$
- Total running time: $O(|V| + |E|)$ (linear in the size of the adjacency list representation of G)

EXERCISE SL09-5

Illustrate DFS on the following graph (start at q). Vertices are considered alphabetically. Adjacency lists are ordered alphabetically. Show the times computed by DFS. In which sequence are the nodes visited?

DFS PROPERTIES/2

DFS versus BFS/1

- BFS visits **all vertices that are reachable** from s
  - returns a single search tree
  - might not visit all nodes
  - considers all directions (paths)
  - finds optimal (shortest) path
  - expensive (chess)
  - is used selectively (on small subproblems)
  - $O(V + E)$

DFS PROPERTIES/2

DFS versus BFS/1

- BFS visits **all vertices that are reachable** from s
  - returns a single search tree
  - might not visit all nodes
  - considers all directions (paths)
  - finds optimal (shortest) path
  - expensive (chess)
  - is used selectively (on small subproblems)
  - $O(V + E)$
- DFS visits **all vertices** in the given graph G
  - may return several search trees
  - visits all nodes
  - pursues one direction (path)
  - can get "lost" (maze)
  - cheap and practical
  - $O(V + E)$

DFS PROPERTIES/3

DFS versus BFS/2

- The difference (visiting all nodes versus visiting reachable nodes) comes from the applications that make use of BFS and DFS

DFS PROPERTIES/3

DFS versus BFS/2

- The difference (visiting all nodes versus visiting reachable nodes) comes from the applications that make use of BFS and DFS
- Note: the behavior of the graph search algorithms (i.e., to search the entire graph or only part of it) could easily be changed

DFS PROPERTIES/4

Parenthesis theorem

- Start and end times have parenthesis structure
  - Represent start time of v with left parenthesis "(v"
  - Represent end time of v with right parenthesis "v)"
  - History of start and end times yields a well-formed expression (i.e. parethesis are properly nested)

DFS PROPERTIES/4

Parenthesis theorem

- Start and end times have parenthesis structure
  - Represent start time of v with left parenthesis "(v"
  - Represent end time of v with right parenthesis "v)"
  - History of start and end times yields a well-formed expression (i.e. parethesis are properly nested)
- Intuition for the proof of the parenthesis theorem
  - Any two intervals are either disjoint or enclosed
  - Overlapping intervals would correspond to one of the following situations
    - Finishing the ancestor before finishing the descendant
    - Starting the descendant before starting the ancestor

DFS PROPERTIES/5

Edge classification/1

- Tree edge (gray node to white node)
  - Edges in depth-first (DF) forest

DFS PROPERTIES/5

Edge classification/1

- Tree edge (gray node to white node)
  - Edges in depth-first (DF) forest
- Back edge (gray node to gray node)
  - Edges from descendant to ancestor in DF tree
  - Self-loops are a special case of back edges

DFS PROPERTIES/5

Edge classification/1

- Tree edge (gray node to white node)
  - Edges in depth-first (DF) forest
- Back edge (gray node to gray node)
  - Edges from descendant to ancestor in DF tree
  - Self-loops are a special case of back edges
- Forward edge (gray node to black node)
  - Non-tree edges from ancestor to descendant in DF tree

DFS PROPERTIES/5

Edge classification/1

- Tree edge (gray node to white node)
  - Edges in depth-first (DF) forest
- Back edge (gray node to gray node)
  - Edges from descendant to ancestor in DF tree
  - Self-loops are a special case of back edges
- Forward edge (gray node to black node)
  - Non-tree edges from ancestor to descendant in DF tree
- Cross edge (gray node to black node)
  - Remaining edges of graph, i.e., edges between trees or subtrees

EXERCISE SL09-6

Draw the DFS forest with edge types. Choose an alphabetical ordering when you can choose between nodes.

DFS PROPERTIES/6

Edge classification/2

- In DFS, the color of the vertex decides the type of the edge

DFS PROPERTIES/6

Edge classification/2

- In DFS, the color of the vertex decides the type of the edge
- It is thus impossible to distinguish forward and cross edges

DFS PROPERTIES/6

Edge classification/2

- In DFS, the color of the vertex decides the type of the edge
- It is thus impossible to distinguish forward and cross edges
- Most algorithms do not distinguish between these two types

DFS PROPERTIES/6

Edge classification/2

- In DFS, the color of the vertex decides the type of the edge
- It is thus impossible to distinguish forward and cross edges
- Most algorithms do not distinguish between these two types
- Tree and back edges are important (e.g., to detect loops)

Table of contents — SL09

1. Graphs

2. Breadth-first search (BFS)

3. Depth-first search (DFS)

4. Topological sorting
   - Directed acyclic graph (DAG)
   - Topological sorting

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- DAGs are used to indicate precedence among events (x must happen before y)
  - An example would be a parallel execution of code

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- DAGs are used to indicate precedence among events (x must happen before y)
  - An example would be a parallel execution of code
- One gets a total order using topological sorting

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- DAGs are used to indicate precedence among events (x must happen before y)
  - An example would be a parallel execution of code
- One gets a total order using topological sorting

DAG theorem

- A directed graph G is acyclic iff DFS of G has no back edges

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- DAGs are used to indicate precedence among events (x must happen before y)
  - An example would be a parallel execution of code
- One gets a total order using topological sorting

DAG theorem

- A directed graph G is acyclic iff DFS of G has no back edges
- Thus, DFS can be used to check if a graph is a DAG

DAG/2

Proof of theorem

- Suppose there is a back edge (u, v)
  - Then v is an ancestor of u in depth-first forest
  - Hence, there is a path from v to u in the graph
  - Such a path together with (u, v) yields a cycle

DAG/2

Proof of theorem

- Suppose there is a back edge (u, v)
  - Then v is an ancestor of u in depth-first forest
  - Hence, there is a path from v to u in the graph
  - Such a path together with (u, v) yields a cycle
- Suppose there is a cycle c in G
  - Let v be the first vertex in c to be discovered
  - Let u be a predecessor of vertex v in cycle c
  - Upon discovering v the path from v to u is white
  - Visiting this path makes u become a descendant of v
  - It follows that edge (u, v) is a back edge

EXERCISE SL09-7/1

Give an algorithm that checks if a graph G is a DAG.

EXERCISE SL09-7/2

Topological sorting/1

Sorting of DAGs

- Topological sorting is used to sort DAGs

Topological sorting/1

Sorting of DAGs

- Topological sorting is used to sort DAGs
- A topological sort of a DAG is a linear ordering of all its vertices s.t., for any edge (u, v) in the DAG, u appears before v in the ordering

Topological sorting/1

Sorting of DAGs

- Topological sorting is used to sort DAGs
- A topological sort of a DAG is a linear ordering of all its vertices s.t., for any edge (u, v) in the DAG, u appears before v in the ordering

Algo: TopologicalSort(G)

Call DFS(G) to compute v.etime for each v;
As each vertex v is finished:
  insert v at the beginning of a linked list;
Return the linked list of vertices

Topological sorting/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$

Topological sorting/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray

Topological sorting/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray
- So, one can distinguish the following cases
  - $v.col = G$: (u, v) is a back edge (cycle, contradiction)
  - $v.col = B$: v is already finished, so $v.etime < u.etime$
  - $v.col = W$: v becomes a descendant of u, so v will be finished before u and $v.etime < u.etime$

Topological sorting/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray
- So, one can distinguish the following cases
  - $v.col = G$: (u, v) is a back edge (cycle, contradiction)
  - $v.col = B$: v is already finished, so $v.etime < u.etime$
  - $v.col = W$: v becomes a descendant of u, so v will be finished before u and $v.etime < u.etime$
- The definition of topological sort is satisfied

Topological sorting/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray
- So, one can distinguish the following cases
  - $v.col = G$: (u, v) is a back edge (cycle, contradiction)
  - $v.col = B$: v is already finished, so $v.etime < u.etime$
  - $v.col = W$: v becomes a descendant of u, so v will be finished before u and $v.etime < u.etime$
- The definition of topological sort is satisfied

Running time

- Total running time of topological sort is $O(|V| + |E|)$

Topological sorting/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray
- So, one can distinguish the following cases
  - $v.col = G$: (u, v) is a back edge (cycle, contradiction)
  - $v.col = B$: v is already finished, so $v.etime < u.etime$
  - $v.col = W$: v becomes a descendant of u, so v will be finished before u and $v.etime < u.etime$
- The definition of topological sort is satisfied

Running time

- Total running time of topological sort is $O(|V| + |E|)$
- Running time of DFS plus $|V|$ insertions into linked list

EXERCISE SL09-8

Run topological sort on the following directed acyclic graph.

Summary

- Graph algorithms
  - Definition (simple path, cycle, subgraph, connected, directed, etc)
  - Status fields of nodes (with possible additional fields)
- Breadth-first search
  - BFS algorithms and properties
  - Application of BFS (e.g., Erdös number, shortest path)
- Depth-first search
  - DFS algorithm and properties
  - Application of DFS (e.g., cycle detection)
- Topological sort: DAG, sorting

Informatik II

Minimum Spanning Trees and Single Source Shorted Paths
— SL10 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

Table of contents — SL10

1. Weighted graphs

2. Minimum spanning tree (MST)

3. Single-source shortest path (SSP)

Weighted graphs

- Weighted graph
  - Graph $G = (V, E)$ where each edge $(u, v) \in E$ has a weight $w(u, v)$ specifying its cost (or length)
  - In other words, G is endowed with a weight function $w\colon E \to \mathbb{R}$ that assigns real numbers to its edges
  - Note: weighted graphs can be directed or undirected

Weighted graphs

- Weighted graph
  - Graph $G = (V, E)$ where each edge $(u, v) \in E$ has a weight $w(u, v)$ specifying its cost (or length)
  - In other words, G is endowed with a weight function $w\colon E \to \mathbb{R}$ that assigns real numbers to its edges
  - Note: weighted graphs can be directed or undirected
- Weighted algorithms
  - Minimum spanning tree
    - Apply to undirected graphs
    - Prim-Jarnik
  - Single-source shortest path
    - Apply to directed graphs
    - Dijkstra

Table of contents — SL10

1. Weighted graphs

2. Minimum spanning tree (MST)
   Prim-Jarnik algorithm
   Prim-Jarnik example

3. Single-source shortest path (SSP)

Minimum spanning tree/1

Spanning trees

- A spanning tree T of G is a subgraph that contains all the vertices of G and that is a tree (thus, acyclic)

Minimum spanning tree/1

Spanning trees

- A spanning tree T of G is a subgraph that contains all the vertices of G and that is a tree (thus, acyclic)
- A minimum spanning tree of (G, w) is a spanning tree T of G that minimizes $w(T) = \sum_{(u,v)\in T} w(u, v)$

Minimum spanning tree/1

Spanning trees

- A spanning tree T of G is a subgraph that contains all the vertices of G and that is a tree (thus, acyclic)
- A minimum spanning tree of (G, w) is a spanning tree T of G that minimizes $w(T) = \sum_{(u,v)\in T} w(u, v)$

Minimum spanning tree/2

Generic algorithm

- Have to
  make $|V|-1$ choices (edges of the MST) to arrive at solution

Minimum spanning tree/2

Generic algorithm

- Have to
  make $|V|-1$ choices (edges of the MST) to arrive at solution

```
A = ∅;
while A does not form a spanning tree do
    Find edge e that is safe for A;
    A = A ∪ {e}
return A;
```

MINIMUM SPANNING TREE/2

Generic algorithm

- Have to
  make $|V|-1$ choices (edges of the MST) to arrive at solution

```
A = ∅;
while A does not form a spanning tree do
    Find edge e that is safe for A;
    A = A ∪ {e}
return A;
```

- After each choice, get a subproblem that is one vertex smaller than the original problem
  - At each step, want to determine cheaply a **save edge** (i.e., that definitely belongs to an MST)
  - Greedy choice property: locally optimal (i.e., greedy) choice yields a globally optimal solution

Minimum spanning tree/3

Greedy choice/1

- Let $G = (V, E)$ and $S \subseteq V$ be a **cut** of G

Minimum spanning tree/3

Greedy choice/1

- Let $G = (V, E)$ and $S \subseteq V$ be a **cut** of G
- That is, G is split into parts S and $V - S$

Minimum spanning tree/3

Greedy choice/1

- Let $G = (V, E)$ and $S \subseteq V$ be a **cut** of G
- That is, G is split into parts S and $V - S$

MINIMUM SPANNING TREE/3

Greedy choice/1

- Let $G = (V, E)$ and $S \subseteq V$ be a **cut** of G
- That is, G is split into parts S and $V - S$

- (u, v) is a **light edge** if it is a min-weight edge of G that connects S and $V - S$ (e.g. (c, d) is such a light edge)

Minimum spanning tree/4

Greedy choice/2

- In the above situation, if (u, v) is a light edge then it belongs to an MST of G

MINIMUM SPANNING TREE/4

Greedy choice/2

- In the above situation, if (u, v) is a light edge then it belongs to an MST of G
- Proof sketch
  - Assume that (u, v) is light but does not belong to any MST of G
  - Look at a path from u to v in some minimum spanning tree T of G
  - Let (x, y) be the first edge on this path that crosses from S to $V - S$
  - If the edge (x, y) of T is replaced by the edge (u, v), $w(T)$ decreases
  - This contradicts the assumption that T is a minimum spanning tree

EXERCISE SL10-1

Let (u, v) be a minimum weight edge in graph G. Show that (u, v) belongs to some MST of G. Hint: Show that from any MST without (u, v) you can construct a MST with (u, v).

Prim-Jarnik algorithm/1

- Vertex-based algorithm

Prim-Jarnik algorithm/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T

Prim-Jarnik algorithm/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T
- Set A covers part of T that is already computed
  - Uses a $pred$ field to indicate the predecessor

Prim-Jarnik algorithm/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T
- Set A covers part of T that is already computed
  - Uses a $pred$ field to indicate the predecessor
- Annotates all the vertices v outside of the set A
  - $v.k$: minimum weight of an edge that connects vertex v with a vertex in A ($v.k = \infty$ if no such edge exists)

Prim-Jarnik algorithm/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T
- Set A covers part of T that is already computed
  - Uses a $pred$ field to indicate the predecessor
- Annotates all the vertices v outside of the set A
  - $v.k$: minimum weight of an edge that connects vertex v with a vertex in A ($v.k = \infty$ if no such edge exists)
- The algorithm relies on a min-priority queue
  - `init_min_pq`, `min_extract`, and `decrease_key`

Prim-Jarnik algorithm/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T
- Set A covers part of T that is already computed
  - Uses a $pred$ field to indicate the predecessor
- Annotates all the vertices v outside of the set A
  - $v.k$: minimum weight of an edge that connects vertex v with a vertex in A ($v.k = \infty$ if no such edge exists)
- The algorithm relies on a min-priority queue
  - `init_min_pq`, `min_extract`, and `decrease_key`
- The running time of the algorithm is $\Theta(E \log V)$

Prim-Jarnik algorithm/2

The algorithm

Algo: PrimJarnik(G,w,s)

---

foreach $v \in G.V$ **do**
  v.key = ∞;
  v.pred = NIL;
s.key = 0;
InitMinPriorityQueue(PQ,G.V);
while $PQ \neq \emptyset$ **do**
  v = ExtractMin(PQ);
  foreach $u \in v.adj$ **do**
    if $u \in PQ \wedge w(v,u) < u.key$ **then**
      u.key = w(v,u);
      u.pred = v;
      DecreaseKey(PQ,u,w(u,v))

Prim-Jarnik example/1

Prim-Jarnik example/1

Prim-Jarnik example/2

Prim-Jarnik example/2

EXERCISE SL10-2

What happens if AH instead of BC is chosen?

Prim-Jarnik example/3

Prim-Jarnik example/3

Prim-Jarnik example/4

Prim-Jarnik example/4

Prim-Jarnik algorithm/5

Prim-Jarnik algorithm/5

EXERCISE SL10-3/1

Assume a computer network. each computer is directly connected to a small number of other computers. Each direct connection has a bandwidth. Determine an efficient network that ensures any two computers are connected, minimizes the number of direct connections, and maximizes the bandwidth.

EXERCISE SL10-3/2

Prim-Jarnik running time

- $Time = V * T(ExtractMin) + O(E) * T(DecreaseKey)$
- Depends on the implementation of the priority queue:

Q	T(ExtractMin)	T(DecreaseKey)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$

- $E \geq V - 1$, $E \leq V^2$, $E = O(V^2)$
- Binary heap implementation:
  $Time = O(V \log V + E \log V) = O(V^2 \log V) = O(E \log V)$

Table of contents — SL10

1. Weighted graphs

2. Minimum spanning tree (MST)

3. Single-source shortest path (SSSP)
   - Single-source shortest path
   - Bellman-Ford algorithm
   - Dijkstra's algorithm

Single-source shortest path/1

Shortest paths

- Generalize directed graphs to weighted setting

Single-source shortest path/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$

Single-source shortest path/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$
- Definition of weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Single-source shortest path/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$
- Definition of weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of minimum weight (cost)

Single-source shortest path/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$
- Definition of weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of minimum weight (cost)
- Single-source shortest path (SSSP) problem
  - Shortest path from source vertex s to any other vertex
  - Other problems: single pair, all pairs, unweighted (BFS)

Single-source shortest path/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$
- Definition of weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of minimum weight (cost)
- Single-source shortest path (SSSP) problem
  - Shortest path from source vertex s to any other vertex
  - Other problems: single pair, all pairs, unweighted (BFS)

Optimal substructure

- The subpaths of shortest paths are shortest paths

Single-source shortest path/2

Paths and cycles

- Shortest paths cannot have cycles
  - Otherwise, improve shortest path by removing cycles
  - No shortest path can have length greater than $|V| - 1$

Single-source shortest path/2

Paths and cycles

- Shortest paths cannot have cycles
  - Otherwise, improve shortest path by removing cycles
  - No shortest path can have length greater than $|V| - 1$

Relaxation/1

- For each vertex v in the graph, maintain $v.dist$, the estimate of the shortest path from s
  - At the start, $v.dist$ is initialized to ∞ and $s.dist$ to 0

Single-source shortest path/3

Relaxation/2

- Relaxing an edge (u, v) means testing whether one can improve the shortest path to v found so far by going through u

Single-source shortest path/3

Relaxation/2

- Relaxing an edge (u, v) means testing whether one can improve the shortest path to v found so far by going through u

Algo: Relax(u,v,w)

if $v.dist > u.dist + w(u, v)$ **then**
  v.dist = u.dist + w(u,v);
  v.pred = u

Single-source shortest path/3

Relaxation/2

- Relaxing an edge (u, v) means testing whether one can improve the shortest path to v found so far by going through u

Algo: Relax(u,v,w)

if $v.dist > u.dist + w(u, v)$ **then**
  v.dist = u.dist + w(u,v);
  v.pred = u

Single-source shortest path/3

Relaxation/2

- Relaxing an edge (u, v) means testing whether one can improve the shortest path to v found so far by going through u

Algo: Relax(u,v,w)

if $v.dist > u.dist + w(u, v)$ **then**
  v.dist = u.dist + w(u,v);
  v.pred = u

Bellman-Ford algorithm/1

- Solves single-source shortest path problem for the case where edge weights may be negative

Bellman-Ford algorithm/1

- Solves single-source shortest path problem for the case where edge weights may be negative
- Returns boolean value indicating whether there is a negative-weight cycle reachable from source vertex s.
  - If there is such a cycle then no solution exists

Bellman-Ford algorithm/1

- Solves single-source shortest path problem for the case where edge weights may be negative
- Returns boolean value indicating whether there is a negative-weight cycle reachable from source vertex s.
  - If there is such a cycle then no solution exists
- Employs relaxation, progressively decreasing $v.dist$

Bellman-Ford algorithm/1

- Solves single-source shortest path problem for the case where edge weights may be negative
- Returns boolean value indicating whether there is a negative-weight cycle reachable from source vertex s.
  - If there is such a cycle then no solution exists
- Employs relaxation, progressively decreasing $v.dist$
- The running time of the algorithm is $\Theta(|E| \cdot |V|)$

Bellman-Ford algorithm/2

The algorithm

Algo: BellmanFord(G,w,s)

---

foreach $v \in G.V$ **do**
  v.dist = ∞; v.pred = NIL;

for i = *1* **to** $|G.V|$-*1* **do**
  foreach $(u, v) \in G.E$ **do**
    Relax(u,v,w);

foreach $(u, v) \in G.E$ **do**
  if $v.dist > u.dist + w(u, v)$ **then**
    return FALSE;

return TRUE;

Bellman-Ford algorithm/3

An example/1

- Consider the following weighted graph

Bellman-Ford algorithm/3

An example/1

- Consider the following weighted graph

- Relax the edges in the following order

1. (t, x)	2. (t, y)	3. (t, z)	4. (x, t)	5. (y, x)
6. (y, z)	7. (z, x)	8. (z, s)	9. (s, t)	10. (s, y)

Bellman-Ford algorithm/3

An example/1

- Consider the following weighted graph

- Relax the edges in the following order

$$1.\ (t,x) \quad 2.\ (t,y) \quad 3.\ (t,z) \quad 4.\ (x,t) \quad 5.\ (y,x)$$
$$6.\ (y,z) \quad 7.\ (z,x) \quad 8.\ (z,s) \quad 9.\ (s,t) \quad 10.\ (s,y)$$

- Negative-weight cycle: let $w(z,x) = 4$

Bellman-Ford algorithm/4

An example/2

Bellman-Ford algorithm/4

An example/2

Bellman-Ford algorithm/5

An example/3

Bellman-Ford algorithm/5

An example/3

Bellman-Ford algorithm/6

An example/4

SSSP in DAGs

- Finding shortest paths in a DAG is much easier

SSSP in DAGs

- Finding shortest paths in a DAG is much easier
- Topological sort determines order of relaxations
- Algorithm:
  1. sort G topologically
  2. consider each vertex in topological sort order
  3. relax each edge leaving the vertex
- We have one relaxation per edge, which gives a runtime of $O(V + E)$ (V times faster than Bellman-Ford)

SSSP in DAGs

- Finding shortest paths in a DAG is much easier
- Topological sort determines order of relaxations
- Algorithm:
  1. sort G topologically
  2. consider each vertex in topological sort order
  3. relax each edge leaving the vertex
- We have one relaxation per edge, which gives a runtime of $O(V + E)$ (V times faster than Bellman-Ford)
- Note: in a DAG there is no negative-weight cycle

Dijkstra's algorithm/1

- Solves single-source shortest path problem for the case where edge weights are nonnegative (otherwise use Bellman-Ford algorithm)

Dijkstra's algorithm/1

- Solves single-source shortest path problem for the case where edge weights are nonnegative (otherwise use Bellman-Ford algorithm)
- Greedy, similar to Prim-Jarnik algorithm for MST
  - Uses a priority queue, changes $dist$ instead of k

Dijkstra's algorithm/1

- Solves single-source shortest path problem for the case where edge weights are nonnegative (otherwise use Bellman-Ford algorithm)
- Greedy, similar to Prim-Jarnik algorithm for MST
  - Uses a priority queue, changes $dist$ instead of k
- In some respect also similar to breadth-first search
  - $v.dist$ does not change further once the vertex v is removed from the (priority) queue

Dijkstra's algorithm/1

- Solves single-source shortest path problem for the case where edge weights are nonnegative (otherwise use Bellman-Ford algorithm)
- Greedy, similar to Prim-Jarnik algorithm for MST
  - Uses a priority queue, changes $dist$ instead of k
- In some respect also similar to breadth-first search
  - $v.dist$ does not change further once the vertex v is removed from the (priority) queue
- The running time of the algorithm is $\Theta(|E| \lg |V|)$

Dijkstra's algorithm/2

The algorithm

Algo: Dijkstra(G,w,s)

---

foreach $v \in G.V$ **do**
  v.dist = ∞; v.pred = NIL;

s.dist = 0;
InitMinPQ(PQ,G.V);
while $PQ \neq \emptyset$ **do**
  v = ExtractMin(PQ);
  foreach $u \in v.adj$ **do**
    Relax(u,v,w);
    DecreaseKey(PQ,u,u.dist)

Dijkstra's algorithm/3

An example/1

Dijkstra's algorithm/3

An example/1

Dijkstra's algorithm/4

An example/2

Dijkstra's algorithm/4

An example/2

Dijkstra's algorithm/5

An example/3

Dijkstra's algorithm/5

An example/3

EXERCISE SL10-4/1

Arbitrage is the practice of taking advantage of a price differential between two or more markets. An example of arbitrage is to use exchange rates to transform one unit of a currency into more than one unit of the same currency. Consider the exchange rates in the table below. For example, with the given exchange rates one can exchange USDs into Euros, Euros into GBPs and GBPs back into USDs. Indeed, for 100 USD one would get
$100 \cdot (0.64 \cdot 0.79 \cdot 1.98) = 100 \cdot 1.001088 \approx 100.11$ USD.

	USD	GBP	CAD	EUR	AUD
USD	1.00	0.50	1.02	0.64	1.04
GBP	1.98	1.00	2.02	1.26	2.06
CAD	0.98	0.49	1.00	0.62	1.02
EUR	1.57	0.79	1.60	1.00	1.64
AUD	0.96	0.48	0.98	0.61	1.00

EXERCISE SL10-4/2

Model the above arbitrage problem as a general graph problem.
Notice that $a \cdot b > 1$ can be rewritten as $(1/a) \cdot (1/b) < 1$, which in turn can be rewritten as $\log(1/a) + \log(1/b) < 0$.
Give an efficient algorithm to determine whether or not there is a sequence of currencies $c_1, \ldots, c_k$ such that
$xch[c_1, c_2] \cdot \ldots \cdot xch[c_{k-1}, c_k] \cdot xch[c_k, c_1] > 1$.

EXERCISE SL10-4/3

Summary

- **Minimum Spanning Trees** (MST):
  - Prim-Jarnik
    - vertex-based
    - priority queue
- **Single Source Shortest Path** (SSSP):
  - Bellman-Ford
    - works for negative weights
    - can be used to detect negative weight cycles
  - Dijkstra
    - very popular
    - greedy
    - priority queue