INFO 6205
Program Structure and Algorithms

Nik Bear Brown

Big-O

Data Structures

Algorithms

Topics

- Big-O
- Data Structures
- Algorithms

Big- O

- $O(g)$
  - the set of functions that grow no faster than g.
- $g(n)$ describes the worst case behavior of an algorithm that is $O(g)$
- Two additional notations
- $\Omega(g)$
  - the set of functions, f, such that

$$f(n) > c\, g(n)$$

for some constant, c, and $n > N$

Big- O

- *Informally,* Time to solve a problem of size, n,

$$T(n) \quad \text{is} \quad O(\log n)$$

➧ $T(n) = c \log_2 n$

- *Formally:*
  - $O(\, g(n)\,)$ is the set of functions, f, such that

$$f(n) < c\, g(n)$$

for some constant, $c > 0$, and $n > N$

  - Alternatively, we may write

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$$

and say

Properties of the O notation

- Constant factors may be ignored
  - $\forall \; k > 0, \; kf$ is $O(f)$
- Higher powers grow faster
  - n^r is $O(n^s)$ if $0 \leq r \leq s$
- Fastest growing term dominates a sum
  - If f is $O(g)$, then $f + g$ is $O(g)$
    eg $an^4 + bn^3$ is $O(n^4)$
- Polynomial's growth rate is determined by leading term
  - If f is a polynomial of degree d,
    then f is $O(n^d)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(\ b^n\ )$ $\forall$ $b > 1$ *and* $k \geq 0$
    eg n^{20} *is* $O(1.05^n)$
- Logarithms grow more slowly than powers
  - $\log_b n$ is $O(\ n^k)$ $\forall$ $b > 1$ *and* $k > 0$
    eg $\log_2 n$ *is* $O(\ n^{0.5})$

Polynomial and Intractable Algorithms

- Polynomial Time complexity
  - An algorithm is said to be polynomial if it is $O(n^d)$ for some integer d
  - Polynomial algorithms are said to be efficient
    - They solve problems in reasonable times!

- Intractable algorithms
  - Algorithms for which there is no *known* polynomial time algorithm
  - *We will come back to this important class later in the course*

A General Portable Performance Metric

- *Informally,* Time to solve a problem of size, n,

$$T(n) \text{ is } O(\log n)$$

➧ $T(n) = c \log_2 n$

- *Formally:*
  - $O(\,g(n)\,)$ is the set of functions, f, such that

$$f(n) < c\,g(n)$$

for some constant, $c > 0$, and $n > N$

ie for sufficiently large ***n***

  - Alternatively, we may write

and say

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$$

g is an upper bound for f

A General Portable Performance Metric

- $O(\, g\,)$
  - the set of functions that grow no faster than g.
- $g(n)$ describes the worst case behaviour of an algorithm that is $O(\, g\,)$
- Two additional notations
- $\Omega(\, g\,)$
  - the set of functions, f, such that

$$f(n) \;>\; c\; g(n)$$

for some constant, c, and $n > N$

g ***is a lower bound for*** ***f***

A General Portable Performance Metric

- $O(g)$
  - the set of functions that grow no faster than g.
- $g(n)$ describes the worst case behavior of an algorithm that is $O(g)$
- Two additional notations
- $\Omega(g)$
  - the set of functions, f, such that

$$f(n) > c\, g(n)$$

for some constant, c, and $n > N$

- $\Theta(g) = O(g) \cap \Omega(g)$

g ***is a lower bound for*** *f*

Set of functions growing at the same rate as *g*

Properties of the O notation

- Constant factors may be ignored
  - $\forall \ k > 0, \ kf$ is $O(f)$

Properties of the O notation

- Constant factors may be ignored
  - $\forall\ k > 0,\ kf$ is $O(f)$
- Higher powers grow faster
  - n^r is $O(n^s)$ if $0 \le r \le s$

Properties of the O notation

- Constant factors may be ignored
  - $\forall\ k > 0,\ kf$ is $O(f)$
- Higher powers grow faster
  - n^r is $O(n^s)$ if $0 \le r \le s$
- Fastest growing term dominates a sum
  - If f is $O(g)$, then $f + g$ is $O(g)$
    $eg\ \ an^4 + bn^3$ is $O(n^4)$

Properties of the O notation

- Constant factors may be ignored
  - $\forall \; k > 0, \; kf$ is $O(f)$
- Higher powers grow faster
  - n^r is $O(n^s)$ if $0 \leq r \leq s$
- Fastest growing term dominates a sum
  - If f is $O(g)$, then $f + g$ is $O(g)$
    $eg \quad an^4 + bn^3$ is $O(n^4)$
- Polynomial's growth rate is determined by leading term
  - If f is a polynomial of degree d,
    then f is $O(n^d)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(b^n) \ \forall \ b > 1 \text{ and } k \geq 0$
    $eg\ n^{20}$ *is* $O(1.05^n)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(b^n)$ $\forall$ $b > 1$ *and* $k \geq 0$
    eg n^{20} *is* $O(1.05^n)$
- Logarithms grow more slowly than powers
  - $\log_b n$ is $O(n^k)$ $\forall$ $b > 1$ *and* $k > 0$
    eg $\log_2 n$ *is* $O(n^{0.5})$

Properties of the O notation

- All logarithms grow at the same rate
  - $\log_b n$ is $O(\log_d n)\ \forall\ \ b, d > 1$

Properties of the O notation

- All logarithms grow at the same rate
  - $\log_b n$ is $O(\log_d n)$ $\forall$ $b, d > 1$

- Sum of first n r^{th} powers grows as the $(r+1)^{th}$ power
  - $\sum_{k=1}^{n} k^r$ is $\Theta(n^{r+1})$

$$eg \quad \sum_{k=1}^{n} i = \frac{n(n+1)}{2} \quad \text{is } \Theta(n^2)$$

Analysing an Algorithm

- Simple statement sequence

  `s1; s2; …. ; sk`

  - $O(1)$ as long as k is constant

- Simple loops

  ```
  for(i=0;i<n;i++) { s; }
  ```

  where `s` is $O(1)$

  - Time complexity is $n\ O(1)$ or $O(n)$

- Nested loops

  ```
  for(i=0;i<n;i++)
      for(j=0;j<n;j++) { s; }
  ```

  - Complexity is $n\ O(n)$ or $O(n^2)$

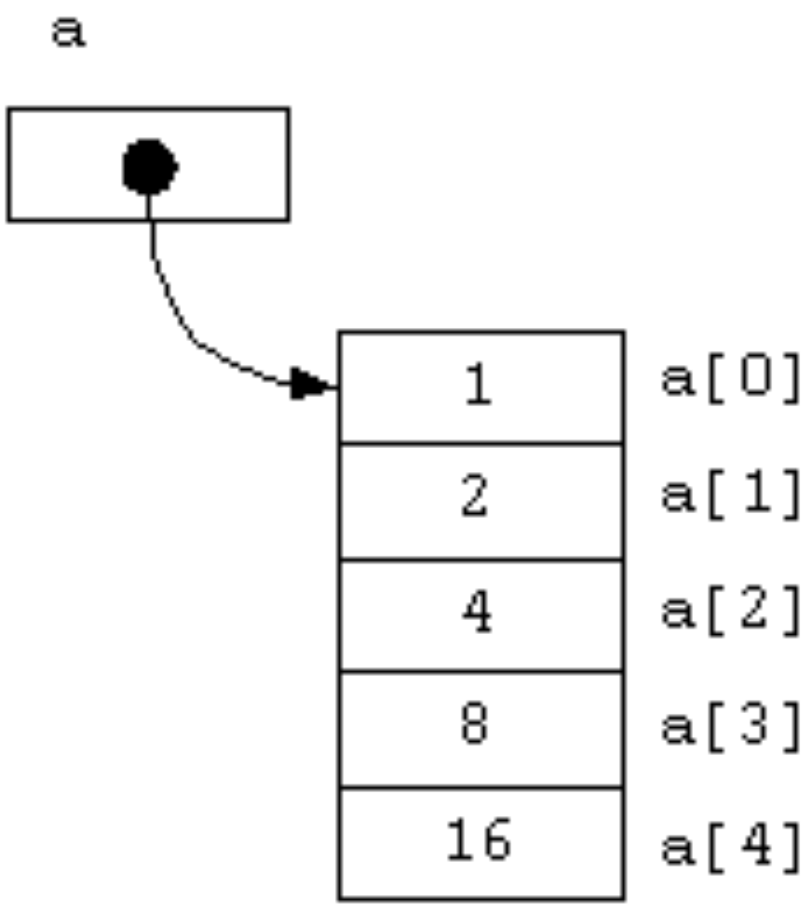
Analysing an Algorithm

- Loop index doesn't vary linearly

```
h = 1;
while ( h <= n ) {
        s;
        h = 2 * h;
        }
```

  - h takes values 1, 2, 4, … until it exceeds *n*
  - There are $1 + \log_2 n$ iterations
  - Complexity $O(\log n)$

Data Structures - Arrays



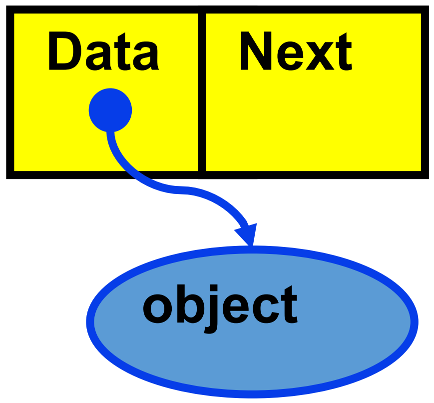
Array Limitations

- Arrays
    - Simple,
    - Fast

    but
    - Must specify size at construction time
    - Murphy’s law
        - Construct an array with space for *n*
            - *n* = twice your estimate of largest collection
        - Tomorrow you’ll need *n*+1
    - More flexible system?

Linked Lists

- Flexible space use
  - Dynamically allocate space for each element as needed
  - Include a pointer to the next item
- Linked list
  - Each node of the list contains
    - the data item (an object pointer in our ADT)
    - a pointer to the next node

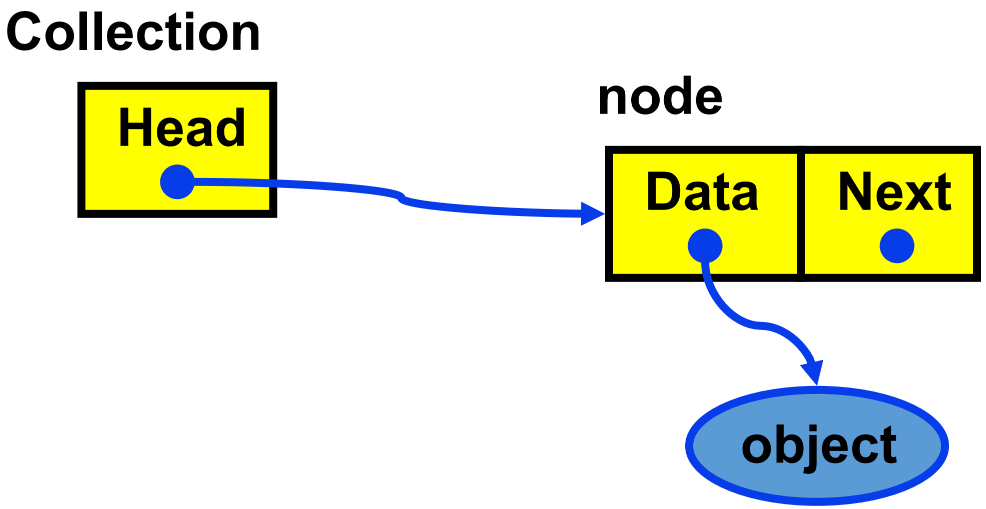


Linked Lists

- Collection structure has a pointer to the list head
  - Initially NULL

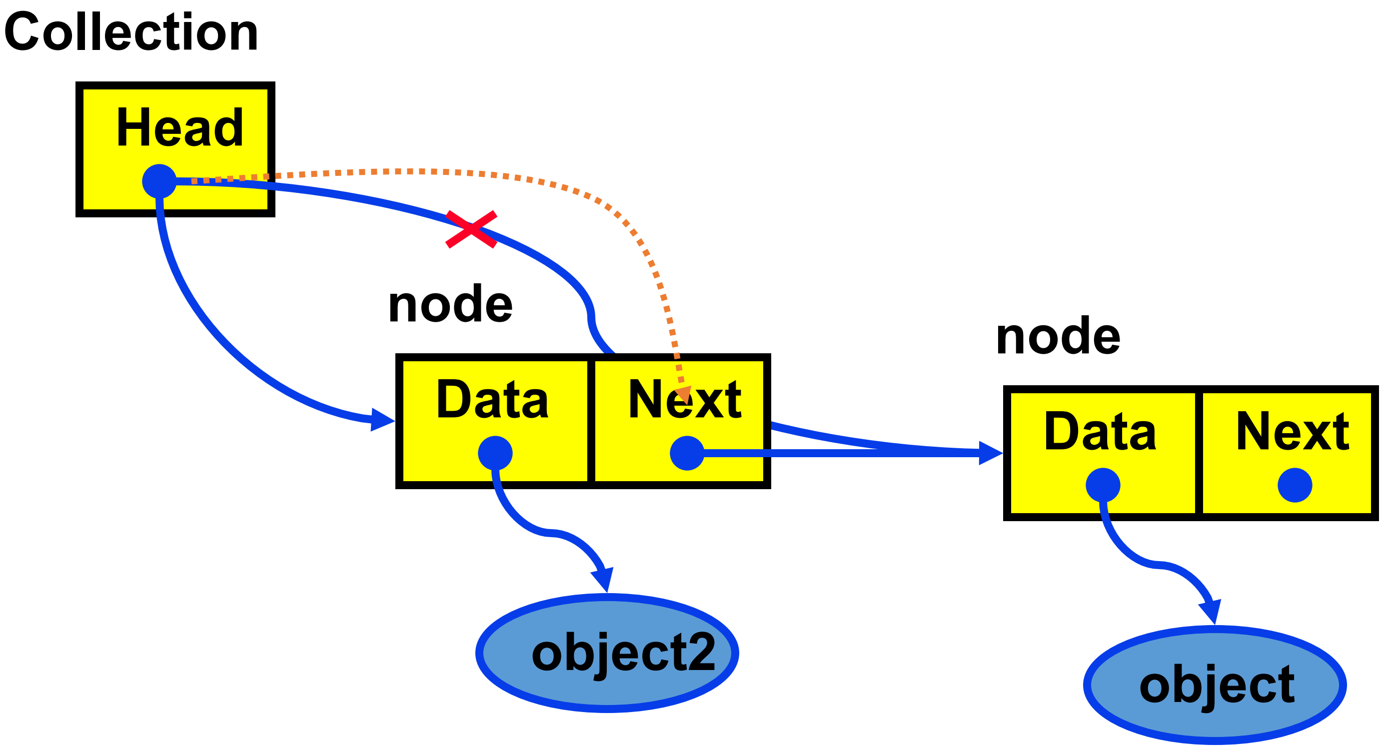
Linked Lists

- Collection structure has a pointer to the list head
  - Initially NULL
- Add first item
  - Allocate space for node
  - Set its data pointer to object
  - Set Next to NULL
  - Set Head to point to new node



Linked Lists

- Add second item
  - Allocate space for node
  - Set its data pointer to object
  - Set Next to current Head
  - Set Head to point to new node



Linked Lists C/C++

```
struct t_node {
    void *item;
    struct t_node *next;
    } node;
typedef struct t_node *Node;
struct collection {
    Node head;
    ……
    };
int AddToCollection( Collection c, void *item ) {
    Node new = malloc( sizeof( struct t_node ) );
    new->item = item;
    new->next = c->head;
    c->head = new;
    return TRUE;
    }
```

Linked Lists - C/C++

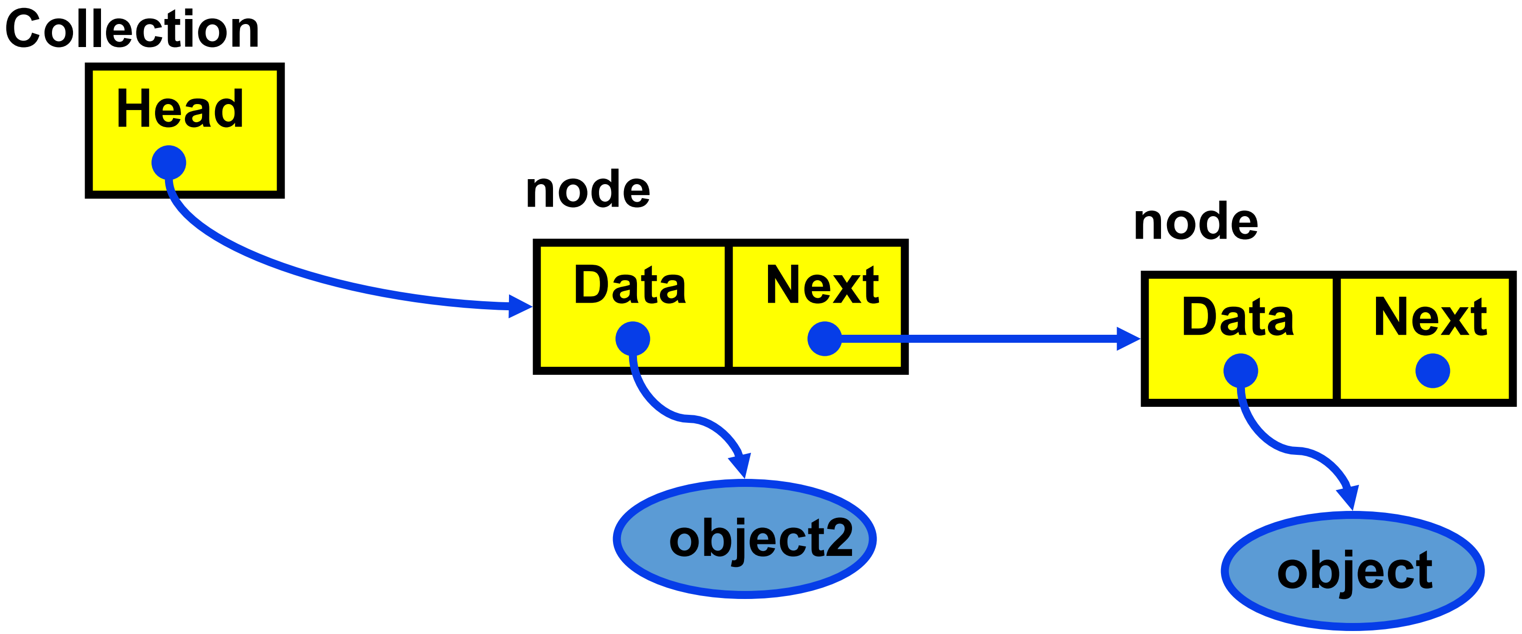
```
struct t_node {
    void *item;
    struct t_node *next;
    } node;
typedef struct t_node *Node;
struct collection {
    Node head;
    ……
    };
int AddToCollection( Collection c, void *item ) {
    Node new = malloc( sizeof( struct t_node ) );
    new->item = item;
    new->next = c->head;
    c->head = new;
    return TRUE;
    }
```

Recursive type definition - C allows it!

Error checking, asserts omitted for clarity!

Linked Lists

- Insertion/Deletion
  - Constant - independent of n
- Search time
  - Worst case - n

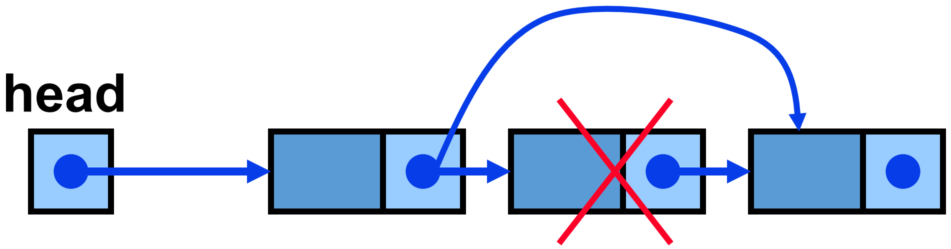


Linked Lists – C/C++

```
void *FindinCollection( Collection c, void *key ) {
    Node n = c->head;
    while ( n != NULL ) {
     if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
           return n->item;
      n = n->next;
      }
    return NULL;
    }
```

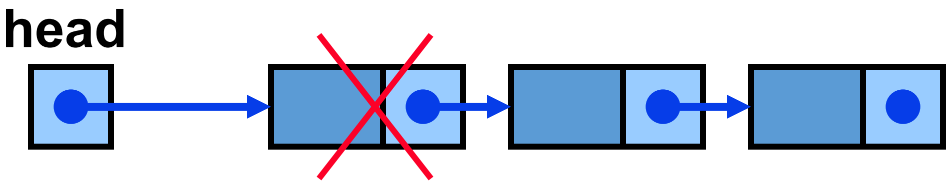
Linked Lists - Delete implementation

```
void *DeleteFromCollection( Collection c, void *key ) {
    Node n, prev;
    n = prev = c->head;
    while ( n != NULL ) {
      if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
            prev->next = n->next;
            return n;
            }
      prev = n;
      n = n->next;
      }
    return NULL;
    }
```



Linked Lists - Delete implementation

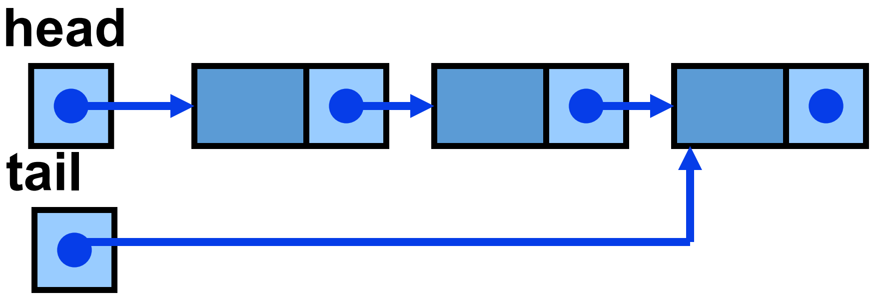
```
void *DeleteFromCollection( Collection c, void *key ) {
    Node n, prev;
    n = prev = c->head;
    while ( n != NULL ) {
      if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
            prev->next = n->next;
            return n;
            }
      prev = n;
      n = n->next;
      }
    return NULL;
    }
```



Minor addition needed to allow for deleting this one! An exercise!

Linked Lists - LIFO and FIFO

- Simplest implementation
  - Add to head
  - ➧ Last-In-First-Out (LIFO) semantics
- Modifications
  - First-In-First-Out (FIFO)
  - Keep a tail pointer



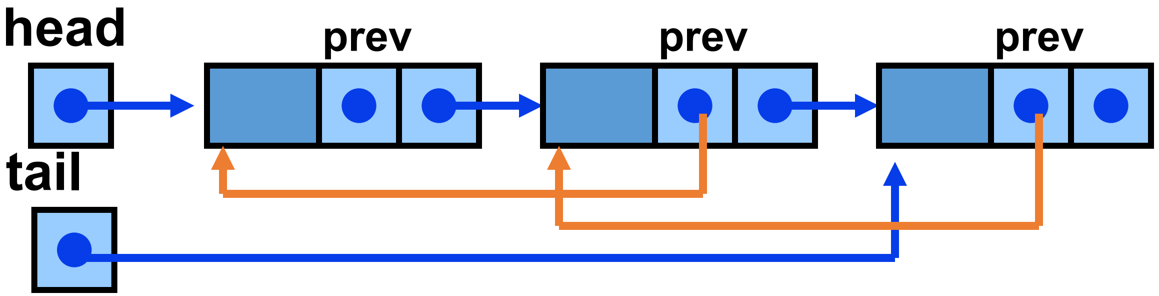

```
struct t_node {
     void *item;
     struct t_node *next;
     } node;
typedef struct t_node *Node;
struct collection {
     Node head, tail;
     };
```

tail is set in the `AddToCollection` method if `head == NULL`

Linked Lists - Doubly linked

- Doubly linked lists
  - Can be scanned in both directions

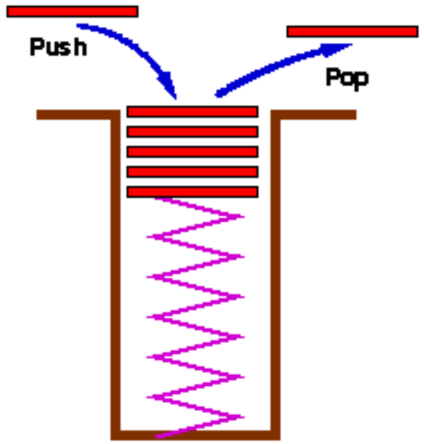
```
struct t_node {
    void *item;
    struct t_node *prev,
                  *next;
    } node;

typedef struct t_node *Node;
struct collection {
    Node head, tail;
    };
```



Stacks

- Stacks are a special form of collection with LIFO semantics
- Two methods
  - int push(Stack s, void *item);
    - add item to the top of the stack
  - void *pop(Stack s);
    - remove an item from the top of the stack
- Like a plate stacker
- Other methods

```
int IsEmpty( Stack s );
/* Return TRUE if empty */
void *Top( Stack s );
/* Return the item at the top,
    without deleting it */
```



Stacks - Implementation

- Arrays
  - Provide a stack capacity to the constructor
  - Flexibility limited *but* matches many real uses
    - Capacity limited by some constraint
      - Memory in your computer
      - Size of the plate stacker, etc
- push, pop methods
  - Variants of AddToC…, DeleteFromC…
- Linked list also possible

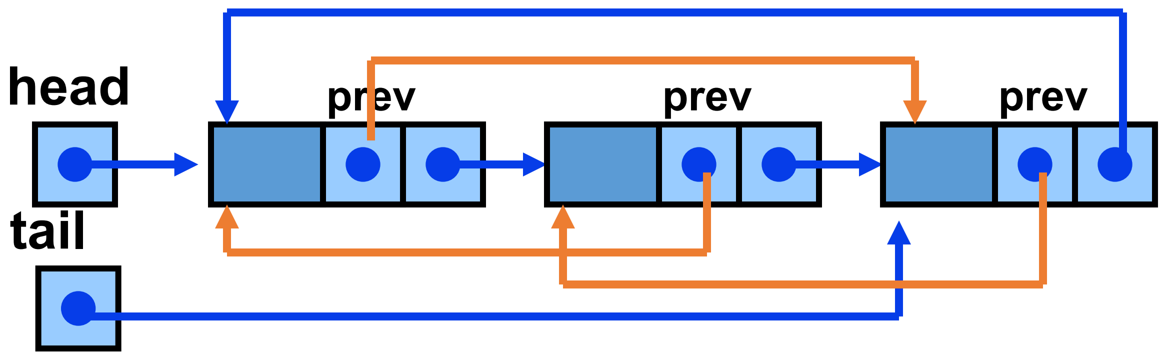
Stacks - Relevance

- Stacks appear in computer programs
  - Key to call / return in functions & procedures
  - Stack frame allows recursive calls
  - Call: push stack frame
  - Return: pop stack frame
- Stack frame
  - Function arguments
  - Return address
  - Local variables

Stacks - Implementation

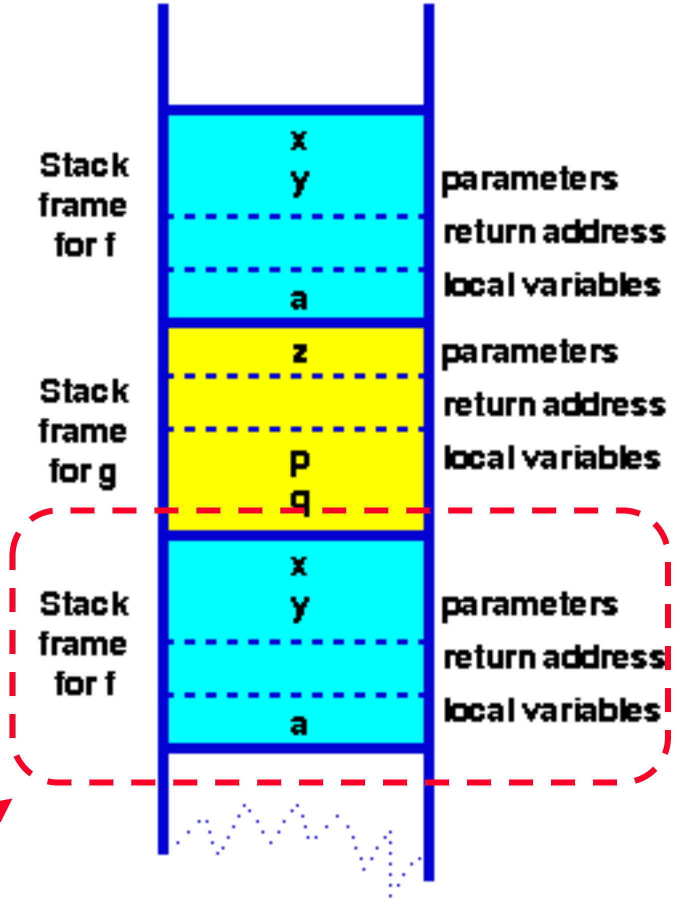
- Arrays common
  - Provide a stack capacity to the constructor
  - Flexibility limited but matches many real uses
    - Stack created with limited capacity

```
struct t_node {
     void *item;
     struct t_node *prev,
                   *next;
     } node;

typedef struct t_node *Node;
struct collection {
     Node head, tail;
     };
```



Stack Frames - Functions in HLL

```
function f( int x, int y) {
   int a;
   if ( term_cond ) return …;
   a = ….;
   return g( a );
   }

function g( int z ) {
   int p, q;
   p = …. ; q = …. ;
   return f(p,q);
   }
```



Context
for execution of `f`

Recursion

- Very useful technique
  - Definition of mathematical functions
  - Definition of data structures
    - Recursive structures are naturally processed by recursive functions!

Recursion

- *Very useful technique*
  - Definition of mathematical functions
  - Definition of data structures
    - Recursive structures are naturally processed by recursive functions!
- Recursively defined functions
  - factorial
  - Fibonacci
  - GCD by Euclid’s algorithm
  - Fourier Transform
  - Games
    - Towers of Hanoi
    - Chess

Recursion - Example

- Fibonacci Numbers

Pseudo-code

```
fib( n ) = if ( n = 0 ) then 1
           else if ( n = 1 ) then 1
           else fib(n-1) + fib(n-2)
```

```
int fib( n ) {
    if ( n < 2 ) return 1;
    else return fib(n-1) +
fib(n-2);
    }
```

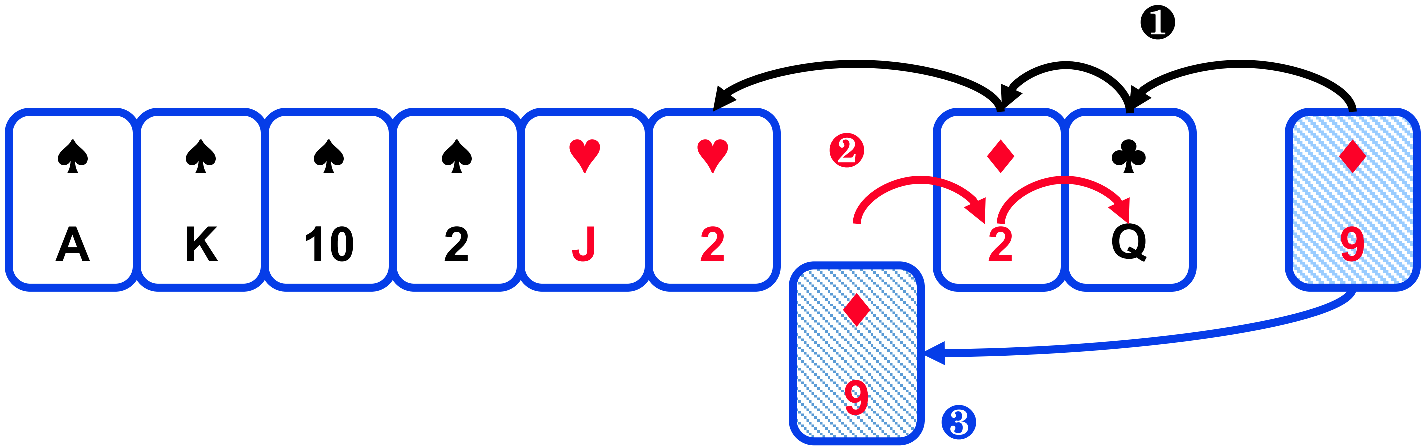
Recursion – Issues?

- Fibonacci Numbers

```
int fib( n ) {
    if ( n < 2 ) return 1;
    else return fib(n-1) + fib(n-2);
    }
```

Sorting

i. Card players all know how to sort …
   i. First card is already sorted
   ii. With all the rest,
      i. Scan back from the end until you find the first card larger than the new one,
      ii. Move all the lower ones up one slot
      iii. insert it



Sorting - Insertion sort

- Complexity
  - For each card
    - Scan $O(n)$
    - Shift up $O(n)$
    - Insert $O(1)$
    - Total $O(n)$

$$\sum_{i=1}^{n} i$$

  - First card requires $O(1)$, second $O(2)$, …
  - For n cards operations ç $O(n^2)$

Sorting - Insertion sort

```
for i ← 1 to length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
```

6 5 3 1 8 7 2 4

Sorting - Insertion sort

```
struct LIST * SortList1(struct LIST * pList) {
    // zero or one element in list
    if(pList == NULL || pList->pNext == NULL)
        return pList;
    // head is the first element of resulting sorted list
    struct LIST * head = NULL;
    while(pList != NULL) {
        struct LIST * current = pList;
        pList = pList->pNext;
        if(head == NULL || current->iValue < head->iValue) {
            // insert into the head of the sorted list
            // or as the first element into an empty sorted list
            current->pNext = head;
            head = current;
        } else {
            // insert current element into proper position in non-empty sorted list
            struct LIST * p = head;
            while(p != NULL) {
                if(p->pNext == NULL || // last element of the sorted list
                   current->iValue < p->pNext->iValue) // middle of the list
                {
                    // insert into middle of the sorted list or as the last element
                    current->pNext = p->pNext;
                    p->pNext = current;
                    break; // done
                }
                p = p->pNext;
            }
        }
    }
    return head;
}
```

Sorting - Insertion sort

Use binary search!

- Complexity
  - For each card
    - Scan $O(n)$ $O(\log n)$
    - Shift up $O(n)$
    - Insert $O(1)$
    - Total $O(n)$
  - First card requires $O(1)$, second $O(2)$, …
  - For n cards operations $\sum_{i=1}^{n} i \Rightarrow O(n^2)$

Unchanged!
Because the
shift up operation
still requires *O(n)*
time

Sorting - Bubble

- From the first element
  - Exchange pairs if they’re out of order
    - Last one must now be the largest
  - Repeat from the first to n-1
  - Stop when you have only one element to check

6 5 3 1 8 7 2 4

Bubble Sort

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

void bubble( int a[], int n ) {
  int i, j;
  for(i=0;i<n;i++) { /* n passes thru the array */
    /* From start to the end of unsorted part */
    for(j=1;j<(n-i);j++) {
      /* If adjacent items out of order, swap */
      if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
      }
    }
  }
```

Bubble Sort - Analysis

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

void bubble( int a[], int n ) {
  int i, j;
  for(i=0;i<n;i++) { /* n passes thru the array */
    /* From start to the end of unsorted part */
    for(j=1;j<(n-i);j++) {
      /* If adjacent items out of order, swap */
      if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
      }
    }
  }
```

***O*(1) statement**

Bubble Sort - Analysis

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

void bubble( int a[], int n ) {
  int i, j;
  for(i=0;i<n;i++) { /* n passes thru the array */
    /* From start to the end of unsorted part */
    for(j=1;j<(n-i);j++) {
      /* If adjacent items out of order, swap */
      if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
      }
    }
  }
```

$O(1)$ statement

Inner loop
n-1, n-2, n-3, … , 1 **iterations**

Bubble Sort - Analysis

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

void bubble( int a[], int n ) {
  int i, j;
  for(i=0;i<n;i++) { /* n passes thru the array */
    /* From start to the end of unsorted part */
    for(j=1;j<(n-i);j++) {
      /* If adjacent items out of order, swap */
      if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
      }
    }
  }
```

Outer loop *n* iterations

Bubble Sort - Analysis

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

void bubble( int a[], int n ) {
  int i, j;
  for(i=0;i<n;i++) { /* n passes thru the array */
    /* From start to the end of unsorted part */
    for(j=
      /* I
      if(
      }
    }
}
```

Overall

$$\sum_{i=n-1}^{1} i = \frac{n(n+1)}{2} = O(n^2)$$

***n* outer loop iterations**

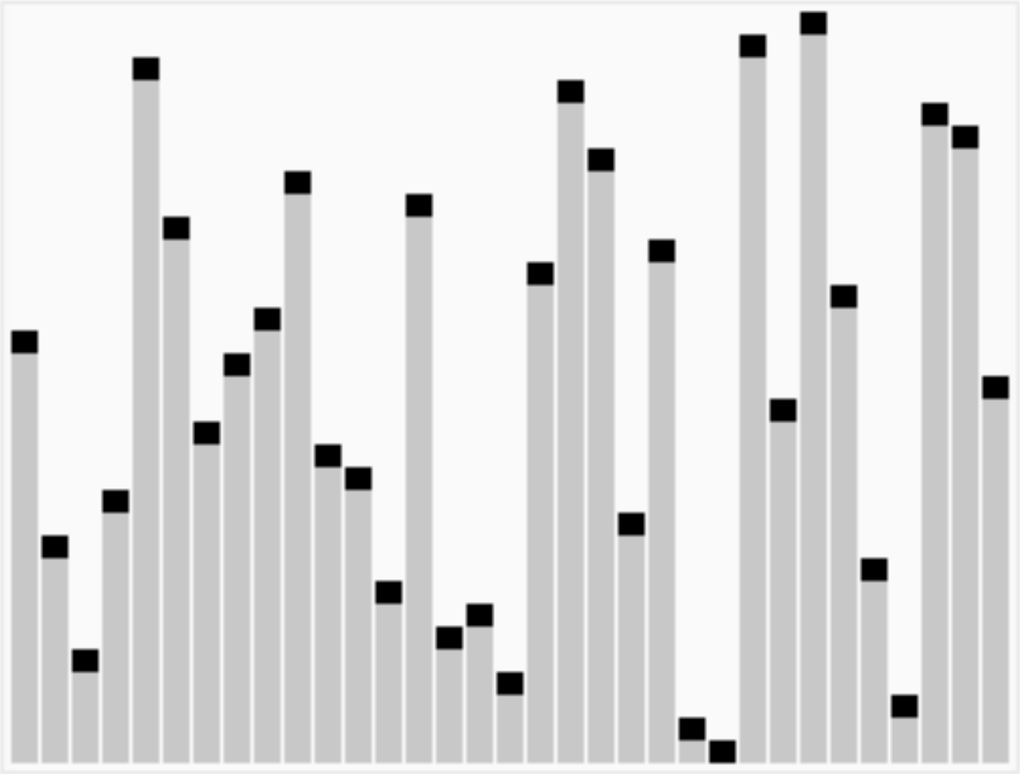
inner loop iteration count

Sorting - Simple

- Bubble sort
  - $O(n^2)$
  - Very simple code
- Insertion sort
  - Slightly better than bubble sort
    - Fewer comparisons
  - Also $O(n^2)$
- *But* HeapSort is $O(n \log n)$
- *Where would you use bubble or insertion sort?*

Quicksort

- Divide and Conquer algorithm
- Two phases
  - Partition phase
    - Divides the work into half
  - Sort phase
    - Conquers the halves!



Quicksort

- Partition
  - Choose a pivot
  - Find the position for the pivot so that
    - all elements to the left are less
    - all elements to the right are greater

< pivot	**pivot**	**> pivot**

Quicksort

- Conquer
  - Apply the same algorithm to each half

Quicksort

```
quicksort( void *a, int low, int high )
  {
  int pivot;
  /* Termination condition! */
  if ( high > low )
    {
    pivot = partition( a, low, high );
    quicksort( a, low, pivot-1 );
    quicksort( a, pivot+1, high );
    }
  }
```

Divide

Conquer

Quicksort - Partition

```
int partition( int *a, int low, int high ) {
   int left, right;
   int pivot_item;
   pivot_item = a[low];
   pivot = left = low;
   right = high;
   while ( left < right ) {
     /* Move left while item < pivot */
     while( a[left] <= pivot_item ) left++;
     /* Move right while item > pivot */
     while( a[right] >= pivot_item ) right--;
     if ( left < right ) SWAP(a,left,right);
     }
   /* right is final position for the pivot */
   a[low] = a[right];
   a[right] = pivot_item;
   return right;
   }
```

Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
      }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
    }
```

This example uses int's to keep things simple!

Any item will do as the pivot, choose the leftmost one!

23	12	15	38	42	18	36	29	27

low

high

Quicksort - Partition

```
int partition( int *a, int low, int high ) {
   int left, right;
   int pivot_item;
   pivot_item = a[low];
   pivot = left = low;
   right = high;
   while ( left < right ) {
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if
      }
   /* right is final position for the pivot */
   a[low] = a[right];
   a[right] = pivot_item;
   return right;
   }
```

Set left and right markers

left | right

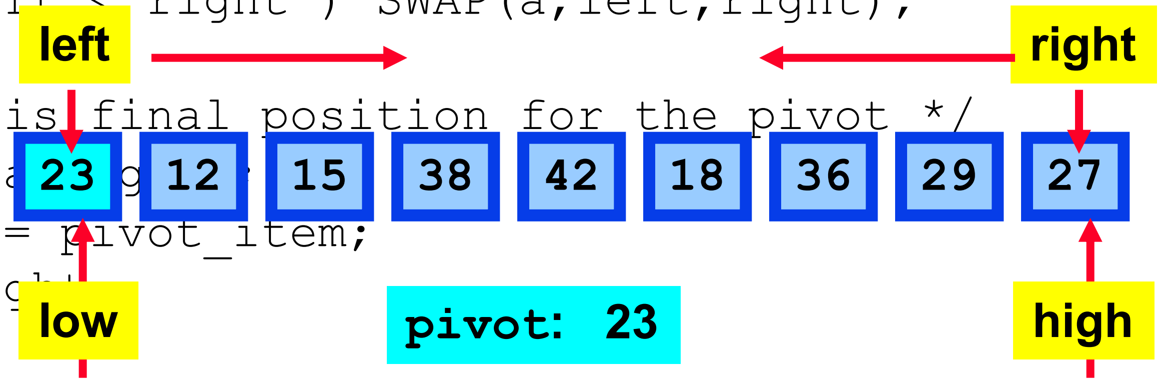
23	12	15	38	42	18	36	29	27

low | pivot: 23 | high

Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
      }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
    }
```

Move the markers until they cross over

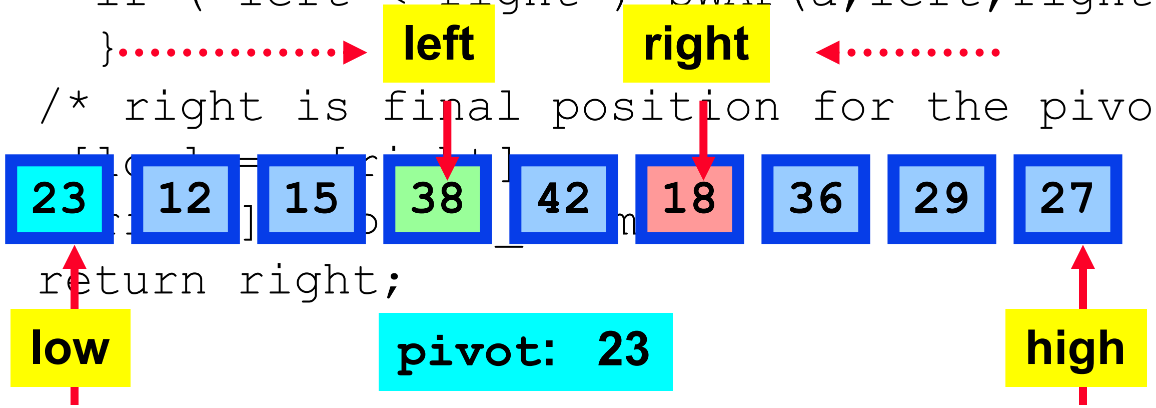


Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
      }
    /* right is final position for the pivot */
    return right;
```

Move the left pointer while it points to items <= pivot

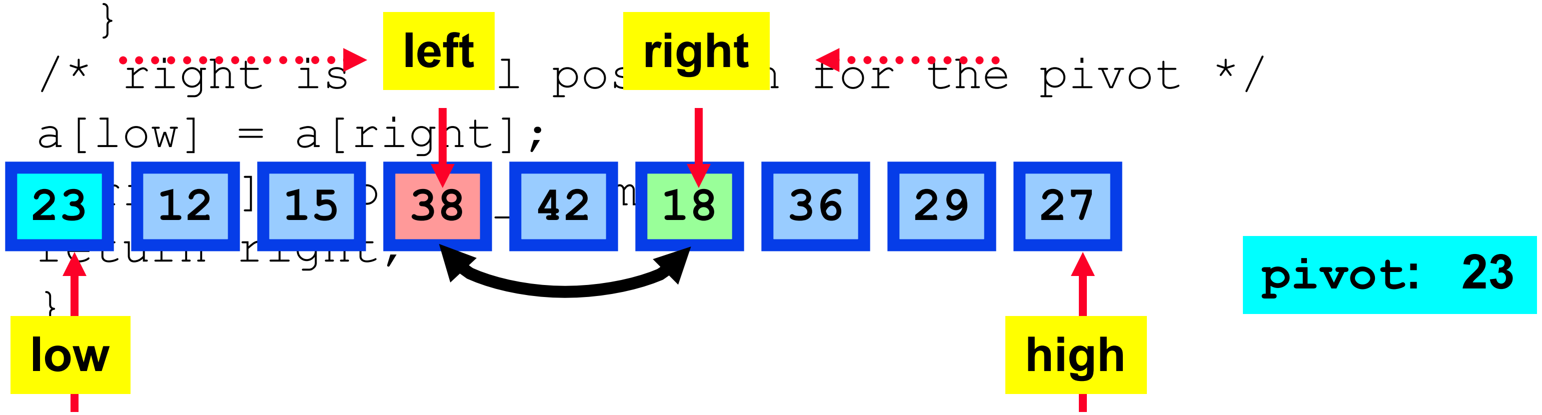
Move right similarly



Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
      }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
    }
```

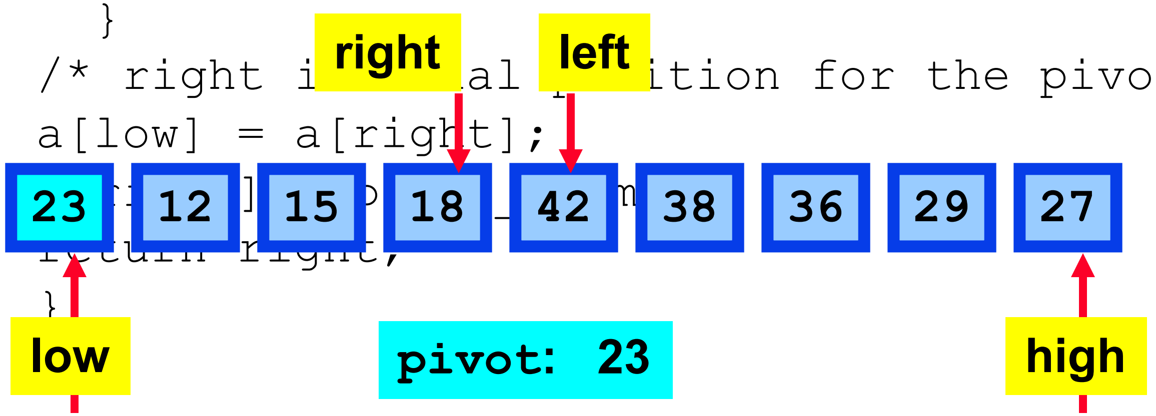
Swap the two items on the wrong side of the pivot



Quicksort - Partition

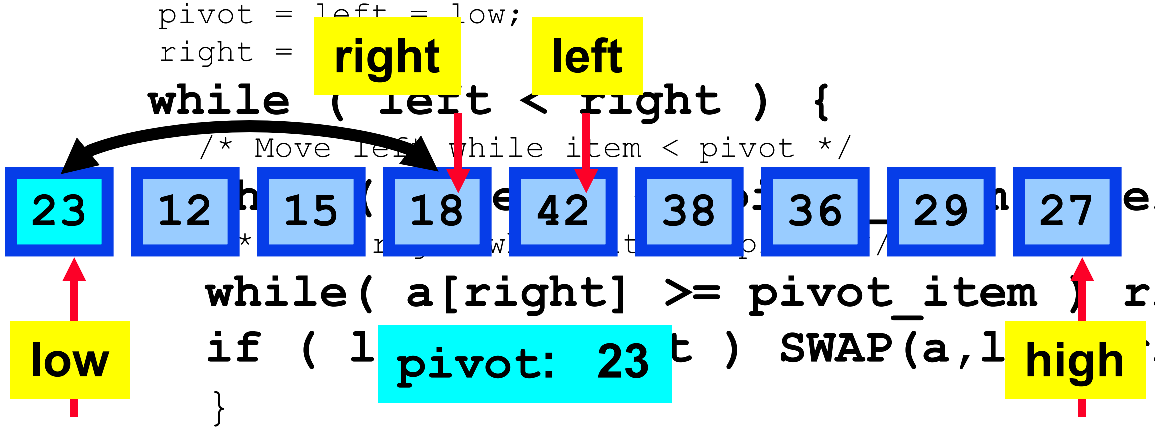
```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
      }
    /* right i   al   ition for the pivot */
    a[low] = a[right];
```

left and right have swapped over, so stop



Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right =
    while ( left < right ) {
      /* Move left while item < pivot */
                                          left++;

      while( a[right] >= pivot_item ) right--;
      if ( l              t ) SWAP(a,l        ight);
      }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
    }
```



Finally, swap the pivot and right

Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = hi
    while ( left < right ) {
      /* Move left while item < pivot */
                                           eft++;
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,l          ight);
      }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_ite
    return right;
    }
```

18	12	15	23	42	38	36	29	27

right

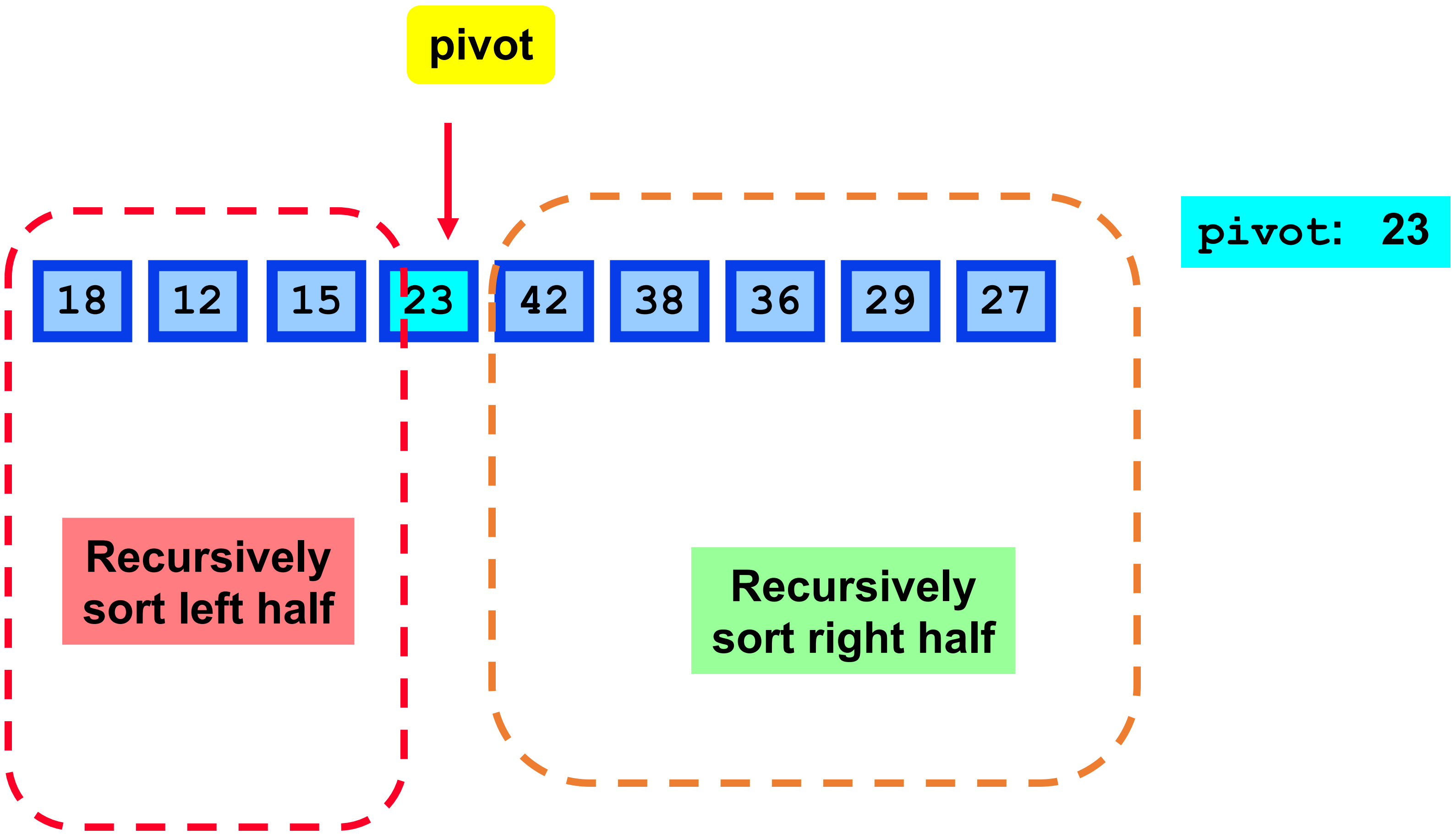
low

high

pivot: 23

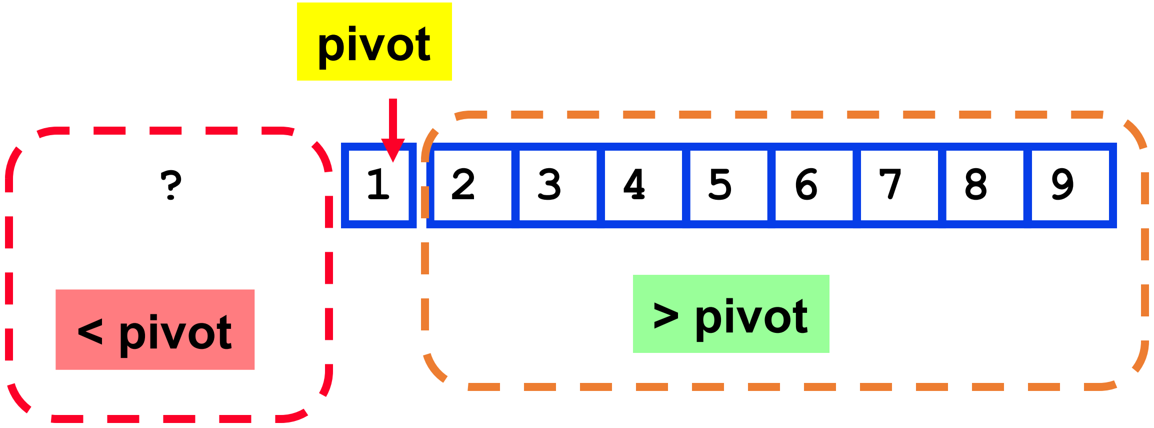
Return the position of the pivot

Quicksort - Conquer



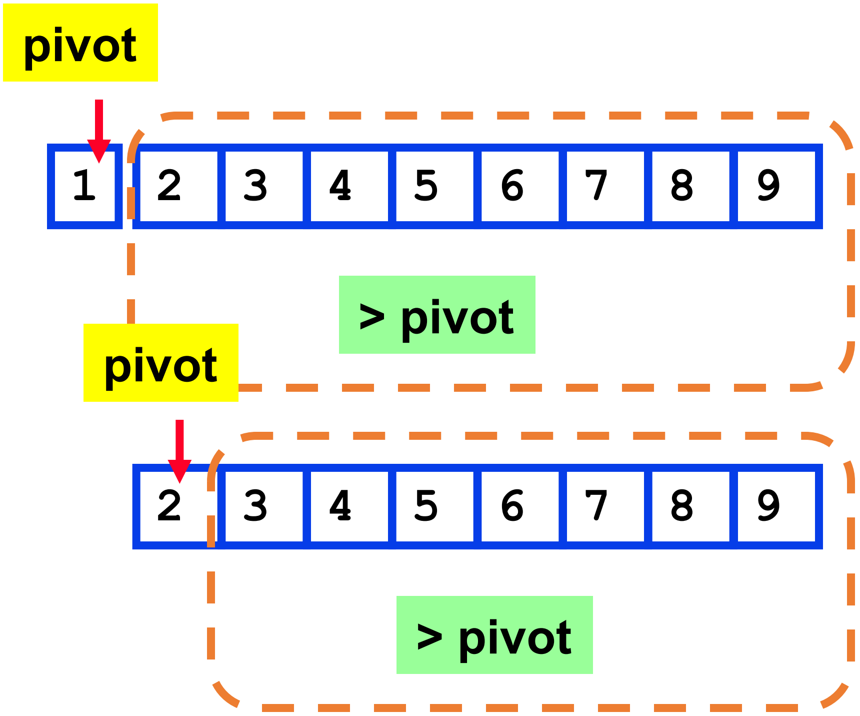
Quicksort

- Sorted data



Quicksort

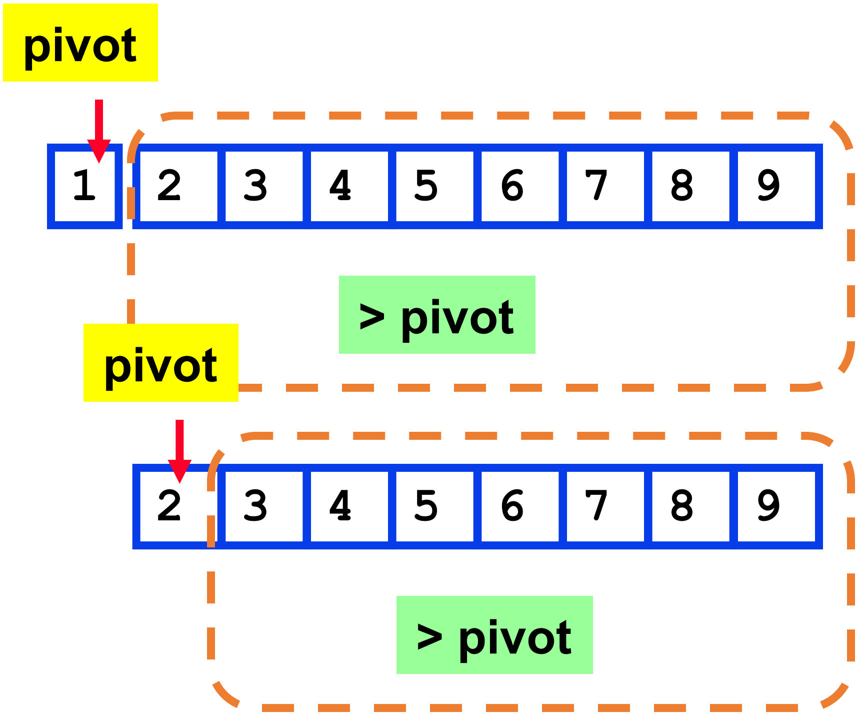
- Sorted data
- Each partition produces
  - a problem of size 0
  - and one of size n-1!
- Number of partitions?



Quicksort

- Sorted data
- Each partition produces
  - a problem of size 0
  - and one of size n-1!
- Number of partitions?
  - n each needing time $O(n)$
  - Total $nO(n)$
    or $O(n^2)$

? *Quicksort is as bad as bubble or insertion sort*



Quicksort

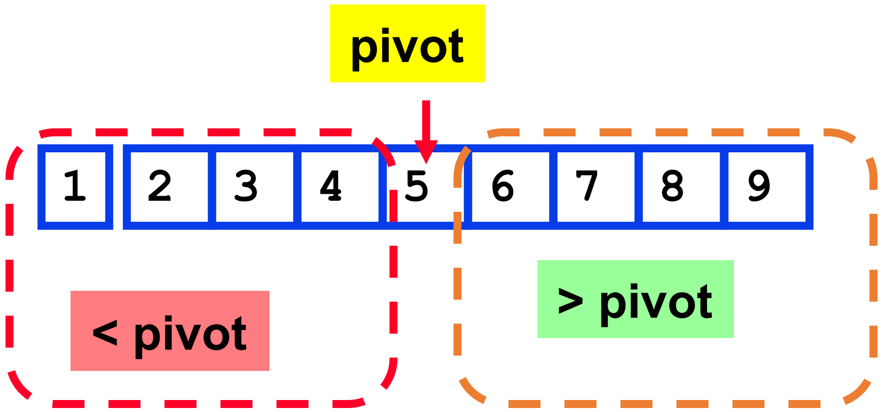
- Quicksort's $O(n \log n)$ behaviour
  - Depends on the partitions being nearly equal
  - ➧ there are $O(log\ n)$ of them
- On average, this will *nearly* be the case

  and quicksort is generally $O(n \log n)$

- Can we do anything to ensure $O(n \log n)$ time?
- In general, no
  - But we can improve our chances!!

Quicksort - Choice of the pivot

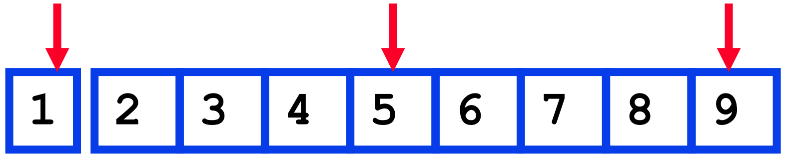
- Any pivot will work …
- Choose a different pivot …



  - so that the partitions are equal
  - *then* we will see $O(n \log n)$ time

Quicksort - Median-of-3 pivot

- Take 3 positions and choose the median
  - say … First, middle, last



- ➧ median is 5
- ➧ perfect division of sorted data every time!
- ➧ *O(n* log *n)* time

➧ Since sorted (or nearly sorted) data is common, median-of-3 is a good strategy

- especially if you think your data may be sorted!

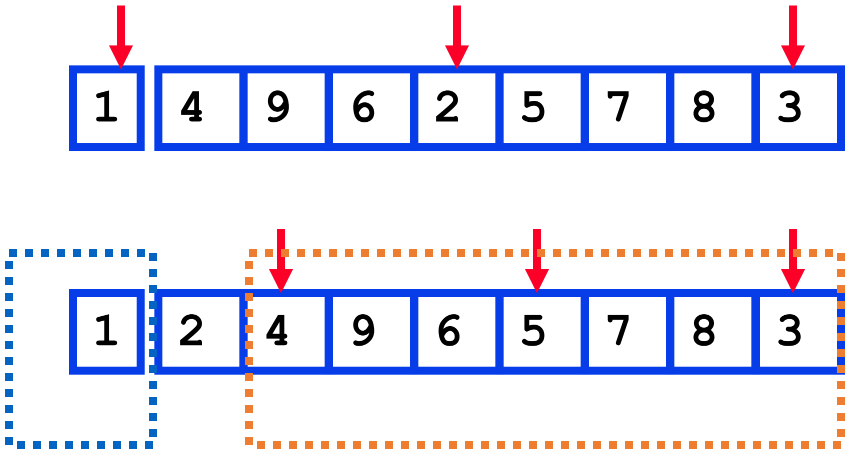
Quicksort - Random pivot

- Choose a pivot randomly
  - Different position for every partition
  - ➧ *On average,* sorted data is divided evenly
  - ➧ $O(n \log n)$ time

- Key requirement
  - Pivot choice must take $O(1)$ time

Quicksort - Guaranteed O(n log n)?

- *Any* pivot selection strategy could lead to $O(n^2)$ time



- Here median-of-3 chooses 2
  - ➡One partition of 1 and
  - One partition of 7
- Next it chooses 4
  - ➡One of 1 and
  - One of 5

Sorting - Key Points

- Sorting
  - Bubble, Insert
    - *$O(n^2)$* sorts
    - Simple code
    - May run faster for small n, n ~10 (system dependent)
  - Quick Sort
    - Divide and conquer
    - $O(n \log n)$

Quicksort - library implementation

```
void qsort( void *base, size_t n, size_t size,
        int (*compar)( const void *, const void * ) );
      base              address of array
      n                 number of elements
      size              size of an element
      compar            comparison function
```

Quicksort - library implementation

```
void qsort( void *base, size_t n, size_t size,
        int (*compar)( const void *, const void * ) );
      base            address of array
      n               number of elements
      size            size of an element
      compar          comparison function
```

Divide-and-Conquer

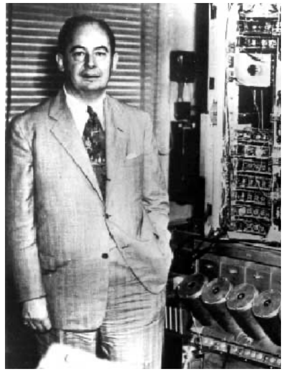
- Divide-and-conquer.
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.

- Most common usage.
  - Break up problem of size n into two equal parts of size ½n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.

- Consequence.
  - Brute force: n^2.
  - Divide-and-conquer: n log n.

Mergesort

- Mergesort.
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.



Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S		
A	L	G	O	R	I	T	H	M	S	divide	O(1)
A	G	L	O	R	H	I	M	S	T	sort	2T(n/2)
A	G	H	I	L	M	O	R	S	T	merge	O(n)

Merging

- Merging. Combine two pre-sorted lists into a sorted whole.

- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.

- Challenge for the bored. In-place merge. [Kronrud, 1969]

A Useful Recurrence Relation

- Def. T(n) = number of comparisons to mergesort an input of size n.

- Mergesort recurrence.

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

- Solution. $T(n) = O(n \log_2 n)$.

- Assorted proofs. We describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace $\le$ with $=$.

Recurrences

- The expression:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

-

  is a *recurrence*.
  - Recurrence: an equation that describes a function in terms of its value on smaller functions

Recurrence Examples

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

The Master Theorem

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size n into a subproblems, each of size n/b
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function $f(n)$
- Then, the Master Theorem gives us a cookbook for the algorithm's running time:

The Master Theorem

- if T(n) = aT(n/b) + f(n) then

$$
T(n) = \begin{cases}
\Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\
\Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\
\Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\
 & af(n/b) < cf(n) \ \text{ for large } n
\end{cases}
\quad
\begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}
$$

Using The Master Method

- $T(n) = 9T(n/3) + n$
  - $a=9$, $b=3$, $f(n) = n$
  - $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
  - Since $f(n) = O(n^{\log_3 9 - \varepsilon})$, where $\varepsilon=1$, case 1 applies:

  - Thus the solution is $T(n) = \Theta(n^2)$

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

$$\text{when} \qquad f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

When Master's Theorem cannot be applied

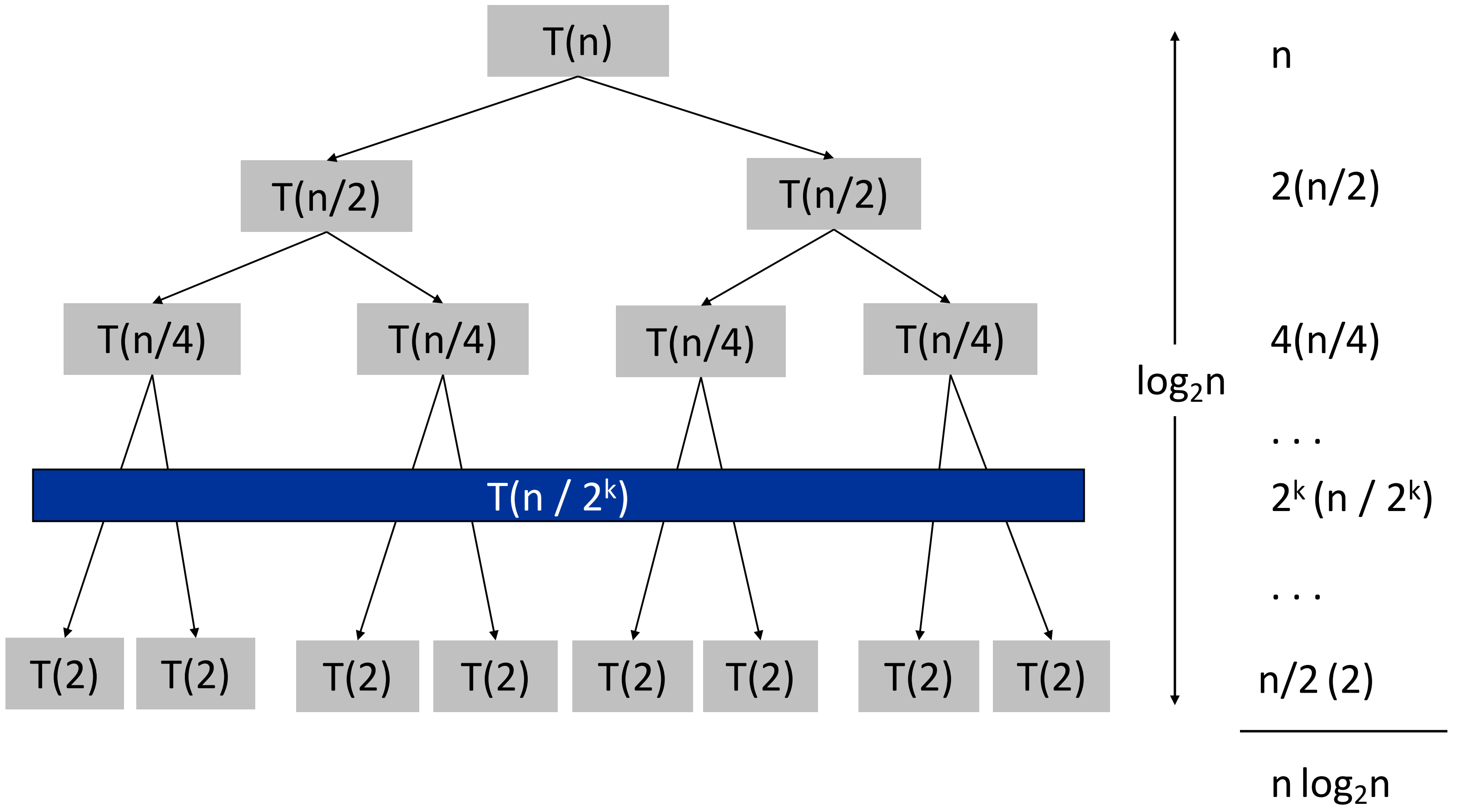
- You cannot use the Master Theorem if
  - $T(n)$ is not monotone, e.g. $T(n) = \sin(x)$
  - $f(n)$ is not a polynomial, e.g., $T(n)=2T(n/2)+2^n$
  - b cannot be expressed as a constant, e.g.

$$T(n) = T(\sqrt{n})$$

- Note that the Master Theorem does not solve the recurrence equation
- Does the base case remain a concern?

Proof by Recursion Tree

$$\mathrm{T}(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Merge Sort Code

```
MergeSort(A, left, right) {
  if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
  }
}
// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A.
// Code for this is in the book.  It requires O(n)
// time, and *does* require allocating O(n) space
```

Analysis of Merge Sort

Statement	Effort
`MergeSort(A, left, right) {`	T(n)
`if (left < right) {`	Θ(1)
`mid = floor((left + right) / 2);`	Θ(1)
`MergeSort(A, left, mid);`	T(n/2)
`MergeSort(A, mid+1, right);`	T(n/2)
`Merge(A, left, mid, right);`	Θ(n)
`}`	
`}`	

- So $T(n) =$ $\Theta(1)$ when $n = 1$, and
  $2T(n/2) + \Theta(n)$ when $n > 1$
- This expression is a *recurrence*

Sorting - Better than O(n log n) ?

- If all we know about the keys is an ordering rule
  - No!
- However,
  - *If we can compute an address from the key*
    (in constant time) then
    bin sort algorithms can provide better performance

Sorting - Bin Sort/ Bucket sort

- Assume
  - All the keys lie in a small, fixed range
    - *eg*
      - integers 0-99
      - characters ‘A’-’z’, ‘0’-’9’
  - There is at most one item with each value of the key
- Bin sort

  Allocate a bin for each value of the key

  Usually an entry in an array

  For each item,

  Extract the key

  Compute it’s bin number

  Place it in the bin

  Finished!

Sorting - Bin Sort/ Bucket sort

```
function bucketSort(array, n) is
  buckets ← new array of n empty lists
  for i = 0 to (length(array)-1) do
    insert array[i] into buckets[msbits(array[i], k)]
  for i = 0 to n - 1 do
    nextSort(buckets[i]);
  return the concatenation of buckets[0], ...., buckets[n-1]!
```

Bin Sort/ Bucket sort Analysis

- All the keys lie in a small, fixed range
  - There are m possible key values
- There is at most one item with each value of the key

- Bin sort

  Allocate a bin for each value of the key $O(m)$

  Usually an entry in an array

  For each item, *n times*

  Extract the key $O(1)$

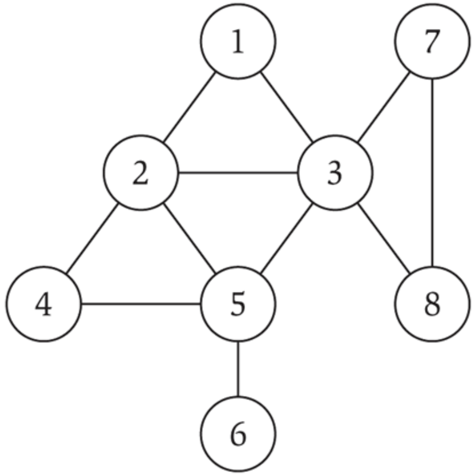
  Compute it's bin number $O(1)$

  Place it in the bin $O(1)$ x n ç $O(n)$

  Finished! $O(n) + O(m) = O(n+m) = O(n)$ *if* $n >> m$

Undirected Graphs

- Undirected graph. G = (V, E)
  - V = nodes.
  - E = edges between pairs of nodes.
  - Captures pairwise relationship between objects.
  - Graph size parameters: n = |V|, m = |E|.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

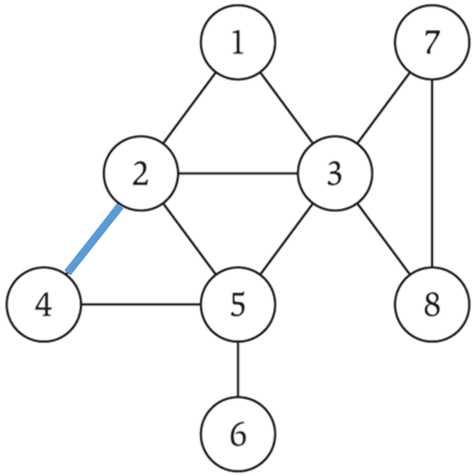
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

m = 11

Graph Representation: Adjacency Matrix

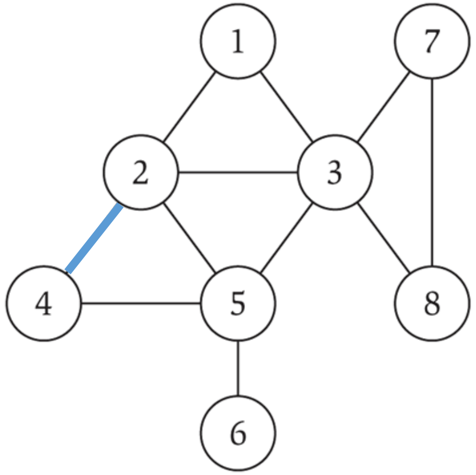
- Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.
  - Two representations of each edge.
  - Space proportional to n^2.
  - Checking if (u, v) is an edge takes $\Theta(1)$ time.
  - Identifying all edges takes $\Theta(n^2)$ time.

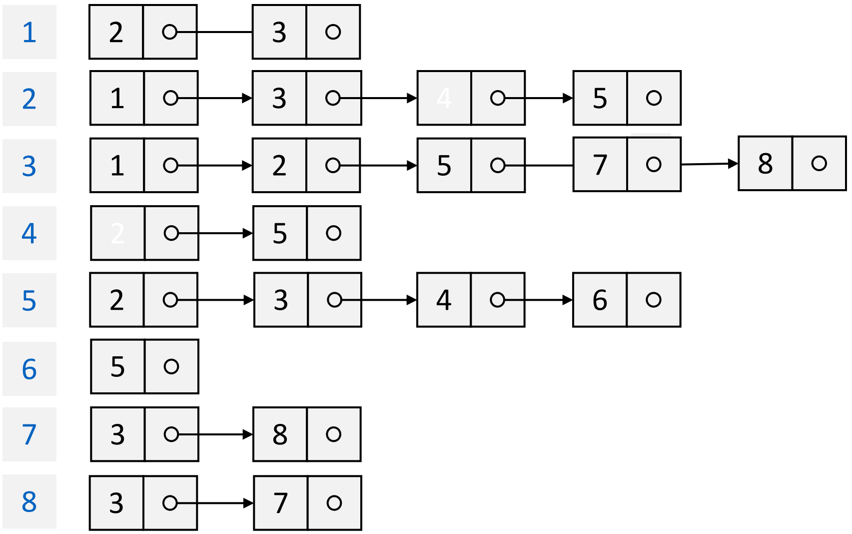


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Representation: Adjacency List

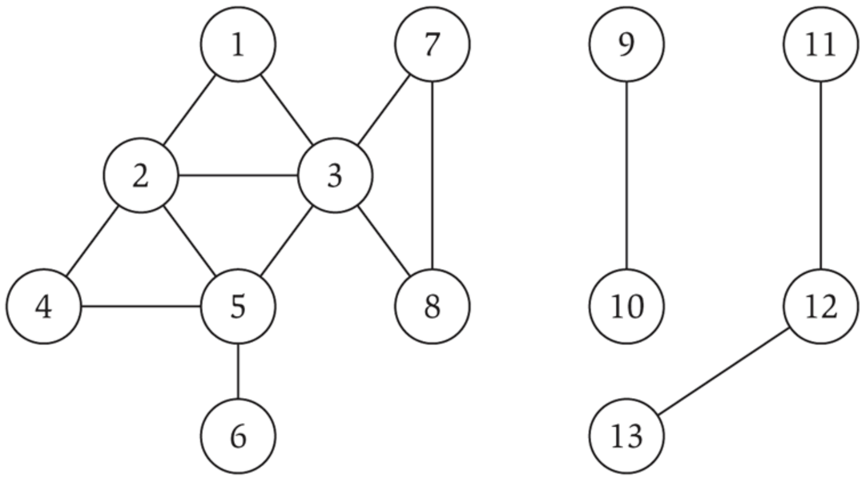
- Adjacency list. Node indexed array of lists.
  - Two representations of each edge.
  - Space proportional to m + n.
  - Checking if (u, v) is an edge takes O(deg(u)) time.
  - Identifying all edges takes Θ(m + n) time.





Paths and Connectivity

- Def. A path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.



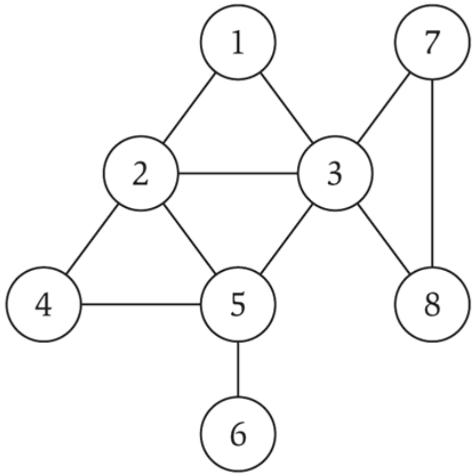
- Def. A path is simple if all nodes are distinct.

- Def. An undirected graph is connected

if for every pair of nodes u and v, there is a path between u and v.

Cycles

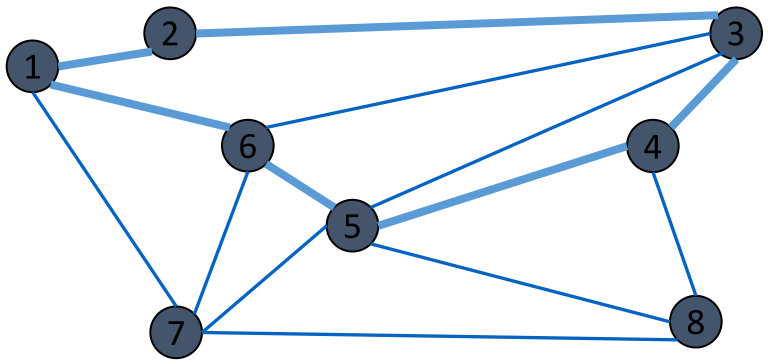
- Def. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct.



cycle C = 1-2-4-5-3-1

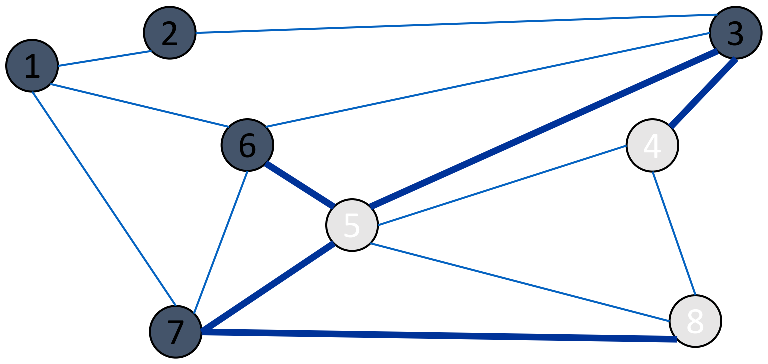
Cycles and Cuts

- Cycle. Set of edges the form a-b, b-c, c-d, …, y-z, z-a.



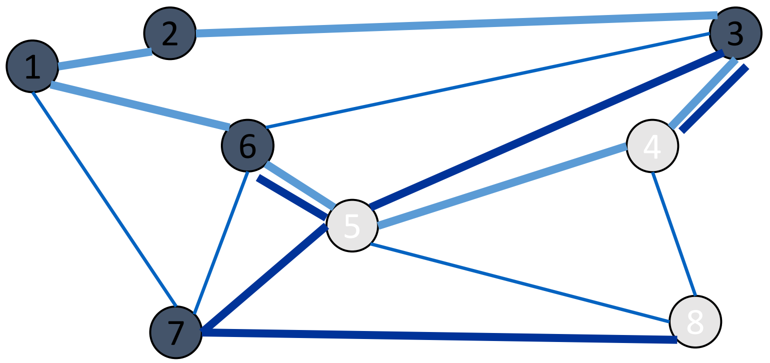
Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

- Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



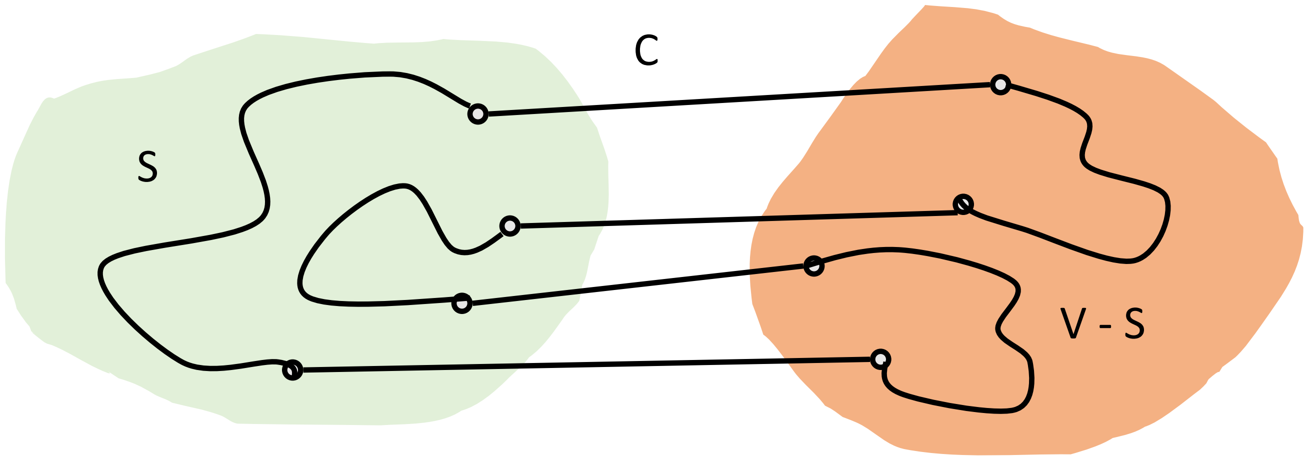
Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

Cycle-Cut Intersection



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

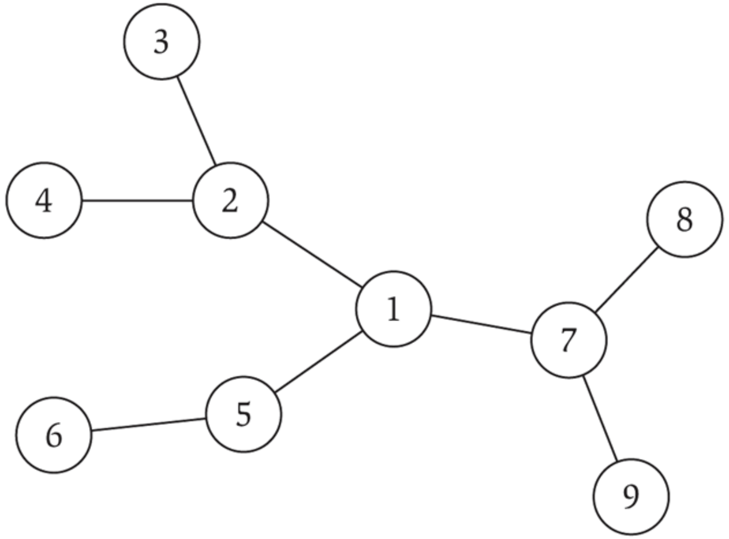
- Claim. A cycle and a cutset intersect in an even number of edges.



Trees

- Def. An undirected graph is a tree if it is connected and does not contain a cycle.

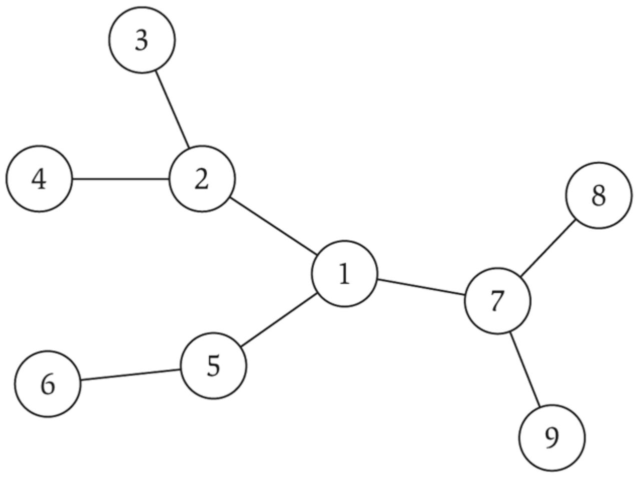
- Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  - G is connected.
  - G does not contain a cycle.
  - G has n-1 edges.

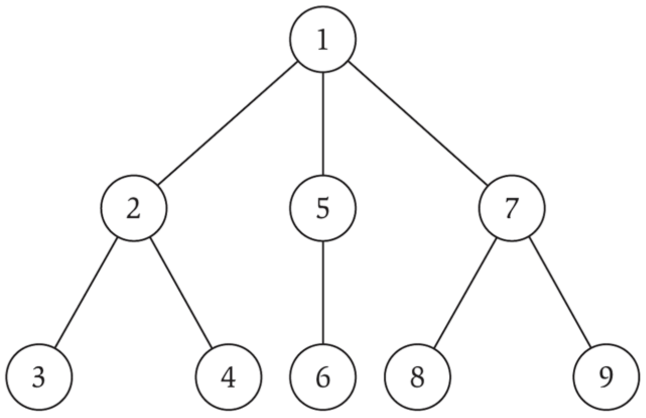


Rooted Trees

- Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.

- Importance. Models hierarchical structure.



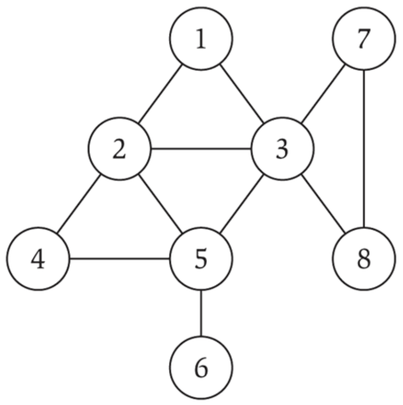


Connectivity

- s-t connectivity problem. Given two node s and t, is there a path between s and t?

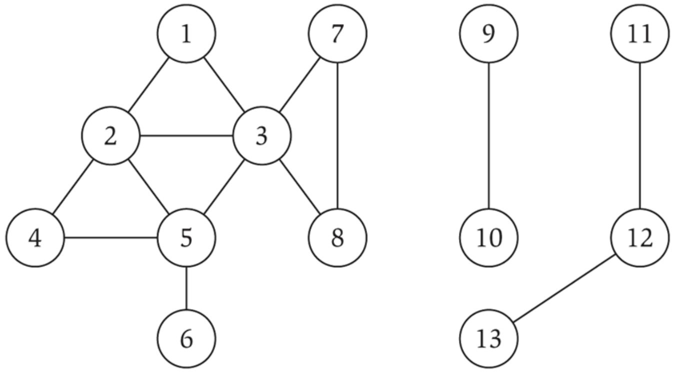
- s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

- Applications.
  - Friendster.
  - Maze traversal.
  - Kevin Bacon number.
  - Fewest number of hops in a communication network.



Connected Component

- Connected component. Find all nodes reachable from s.



- Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

Connected Component

R will consist of nodes to which s has a path

Initially $R = \{s\}$

While there is an edge (u, v) where $u \in R$ and $v \notin R$

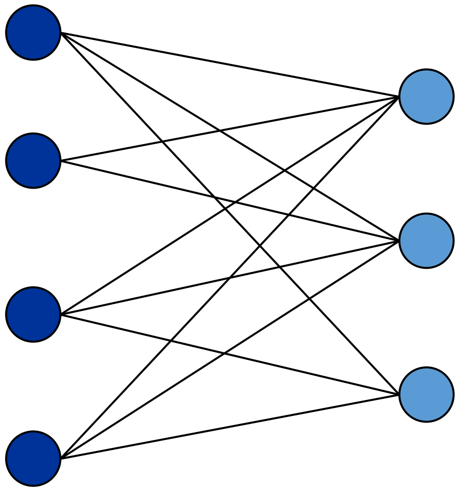
    Add v to R

Endwhile

Bipartite Graphs

- Def. An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

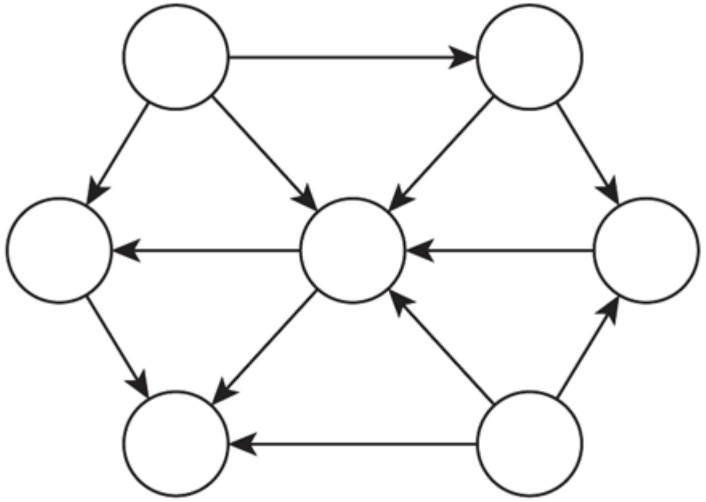
- Applications.
  - Stable marriage: men = red, women = blue.
  - Scheduling: machines = red, jobs = blue.



a bipartite graph

Directed Graphs

- Directed graph. G = (V, E)
  - Edge (u, v) goes from node u to node v.



- Ex. Web graph - hyperlink points from one web page to another.
  - Directedness of graph is crucial.
  - Modern web search engines exploit hyperlink structure to rank web pages by importance.

Graph Search

- Directed reachability. Given a node s, find all nodes reachable from s.

- Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

- Graph search. BFS extends naturally to directed graphs.

- Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

Strong Connectivity

- Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

- Def. A graph is strongly connected if every pair of nodes is mutually reachable.

- Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

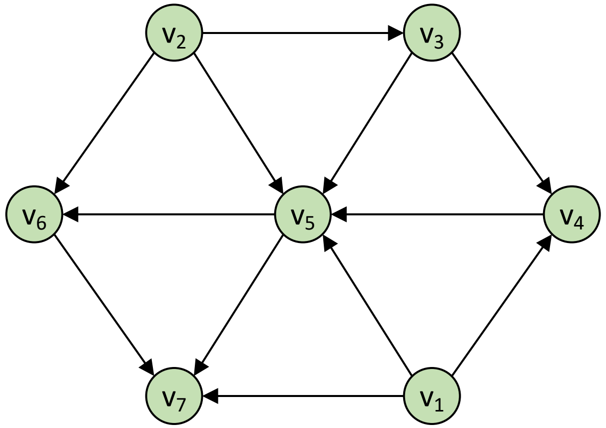
- Pf. $\Rightarrow$ Follows from definition.
- Pf. $\Leftarrow$ Path from u to v: concatenate u-s path with s-v path.
  Path from v to u: concatenate v-s path with s-u path. ▪

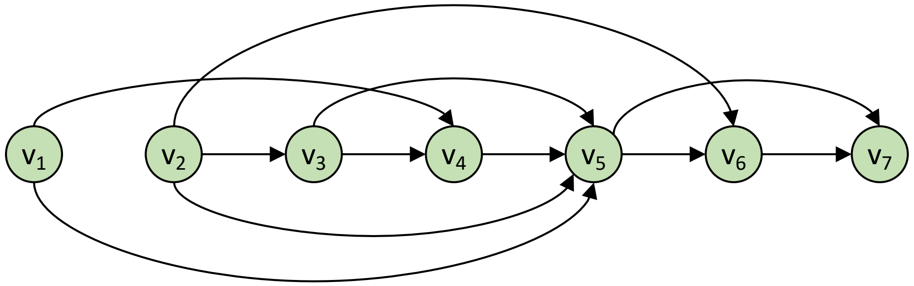
Directed Acyclic Graphs

- Def. An DAG is a directed graph that contains no directed cycles.

- Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

- Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.





Graphs - Traversing

- Choices
  - Depth-First / Breadth-first
- Depth First
  - Use an array of flags to mark "visited" nodes

Graph - Breadth-first Traversal

- Adjacency List
  - Time complexity
    - Visited set for each node
    - Each edge visited twice
      - Once in each adjacency list
    - $O(|V| + |E|)$
    - $O(|V|^2)$ for dense $|E| \sim |V|^2$ graphs
    - *but* $O(|V|)$ for sparse $|E| \sim |V|$ graphs
- Adjacency Lists perform better for sparse graphs

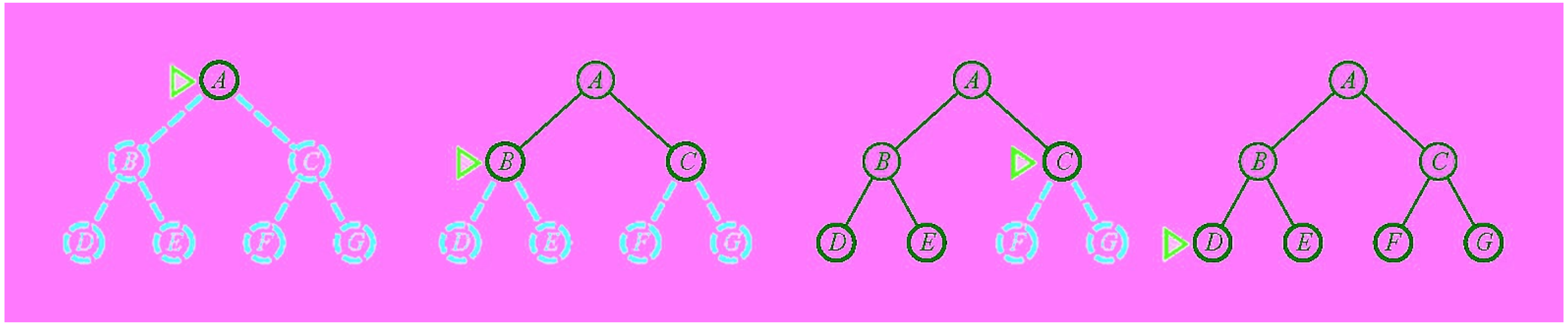
Graph - Breadth-first Traversal

```
static queue q;
void search( graph g ) {
  q = ConsQueue( g->n_nodes );
  for(k=0;k<g->n_nodes;k++) g->visited[k] = 0;
  search_index = 0;
  for(k=0;k<g->n_nodes;k++) {
    if ( !g->visited[k] ) visit( g, k );
  }

void visit( graph g, int k ) {
  al_node al_node;
  int j;
  AddIntToQueue( q, k );
  while( !Empty( q ) ) {
    k = QueueHead( q );
    g->visited[k] = ++search_index;
    ......
```

Graph - Breadth-first Traversal

```
void visit( graph g, int k ) {
  al_node al_node;
  int j;
  AddIntToQueue( q, k );
  while( !Empty( q ) ) {
    k = QueueHead( q );
    g->visited[k] = ++search_index;
    al_node = ListHead( g->adj_list[k]);
    while( al_node != NULL ) {
      j = ANodeIndex(al_node);
      if ( !g->visited[j] ) {
        AddIntToQueue( g, j );
        g->visited[j] = -1; /* C hack, 0 = false! */
        al_node = ListNext( al_node );
        }
      }
    }
  }
```

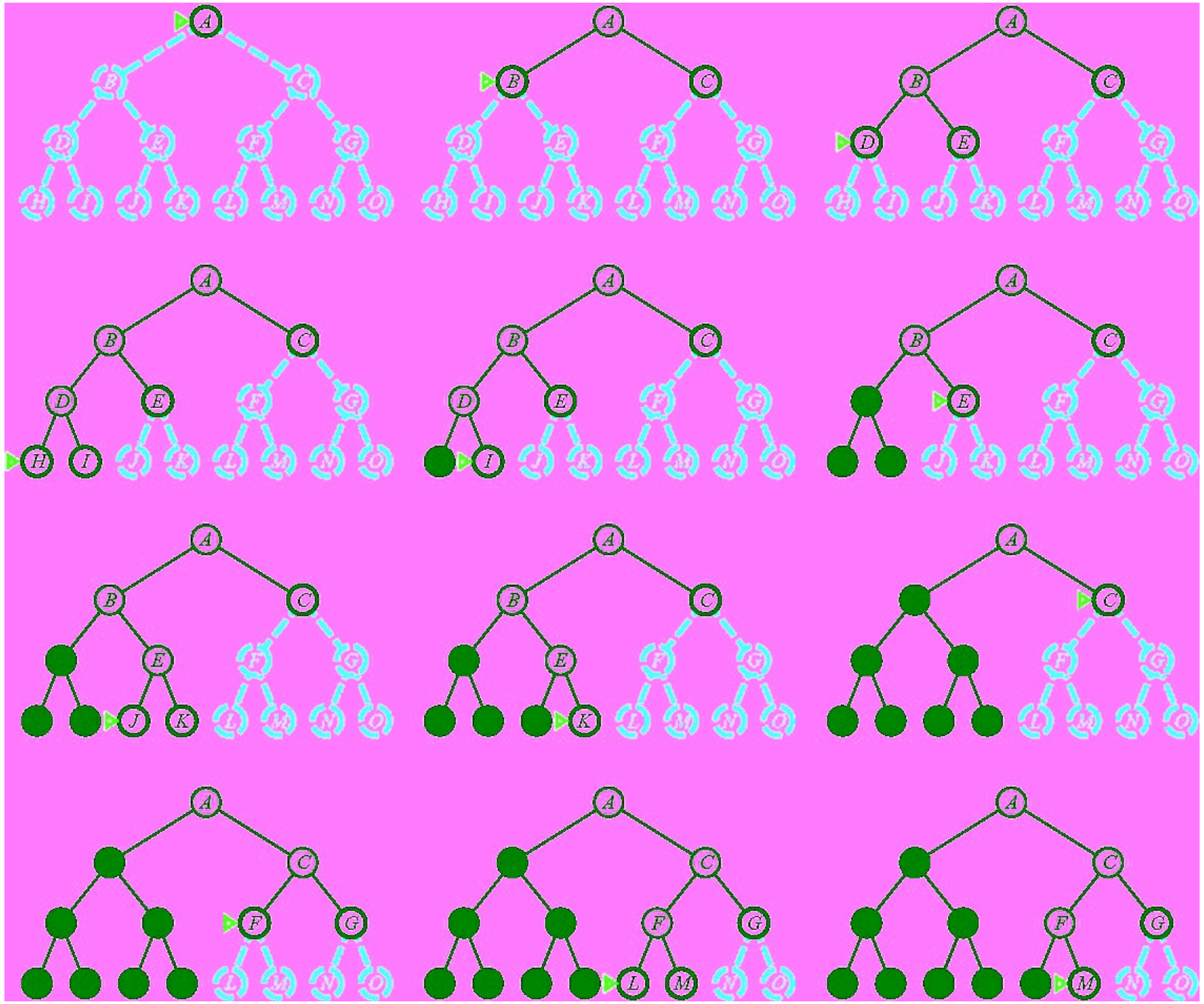
Breadth-First Search



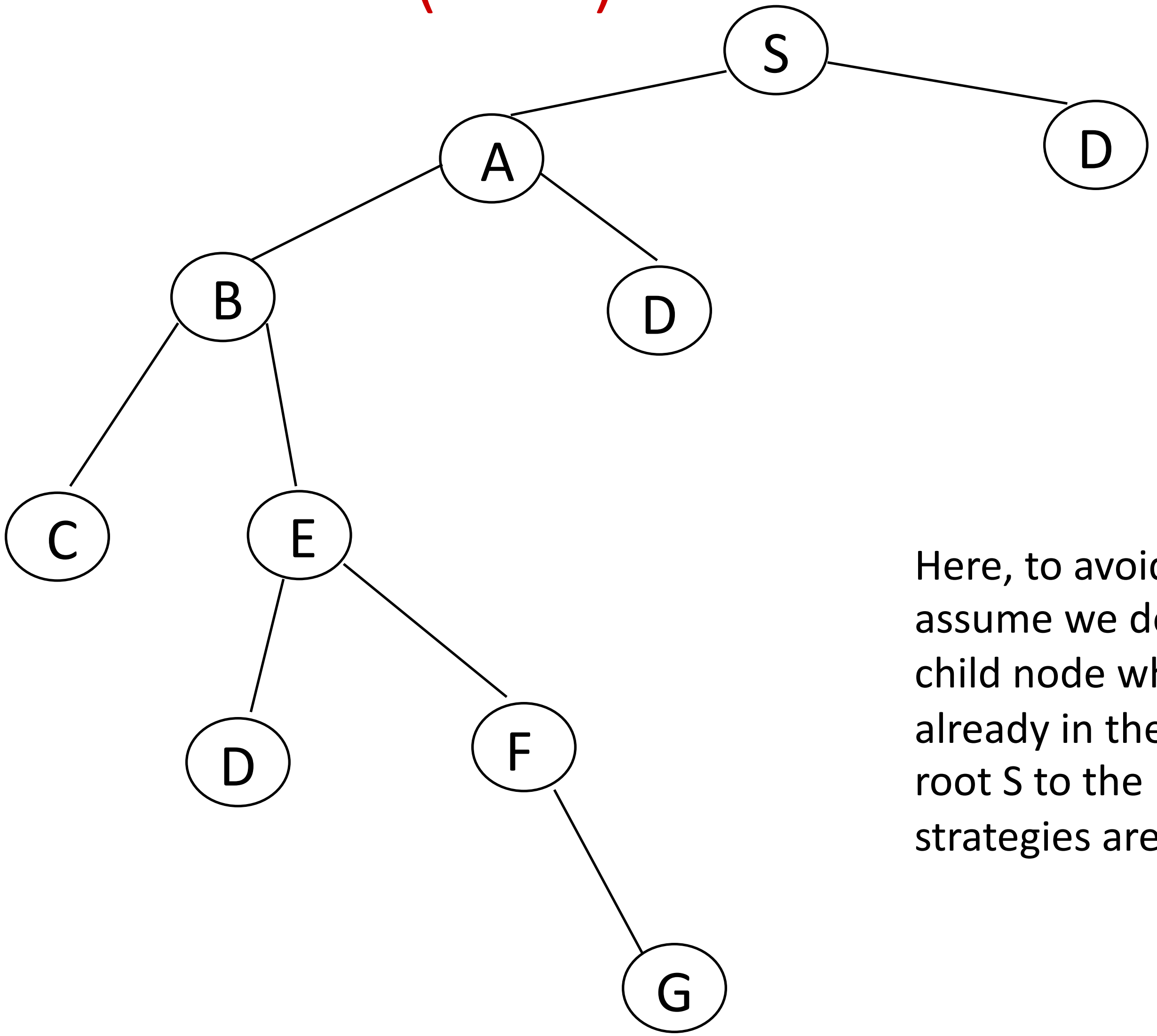
Pseudocode for Breadth-First Search

```
Initialize: Let Q = {S}
While Q is not empty
        pull Q1, the first element in Q
        if Q1 is a goal
                report(success) and quit
        else
                child_nodes = expand(Q1)
                eliminate child_nodes which represent loops
                put remaining child_nodes at the back of Q
        end
Continue
```

Depth-First Search



Depth First Search (DFS)

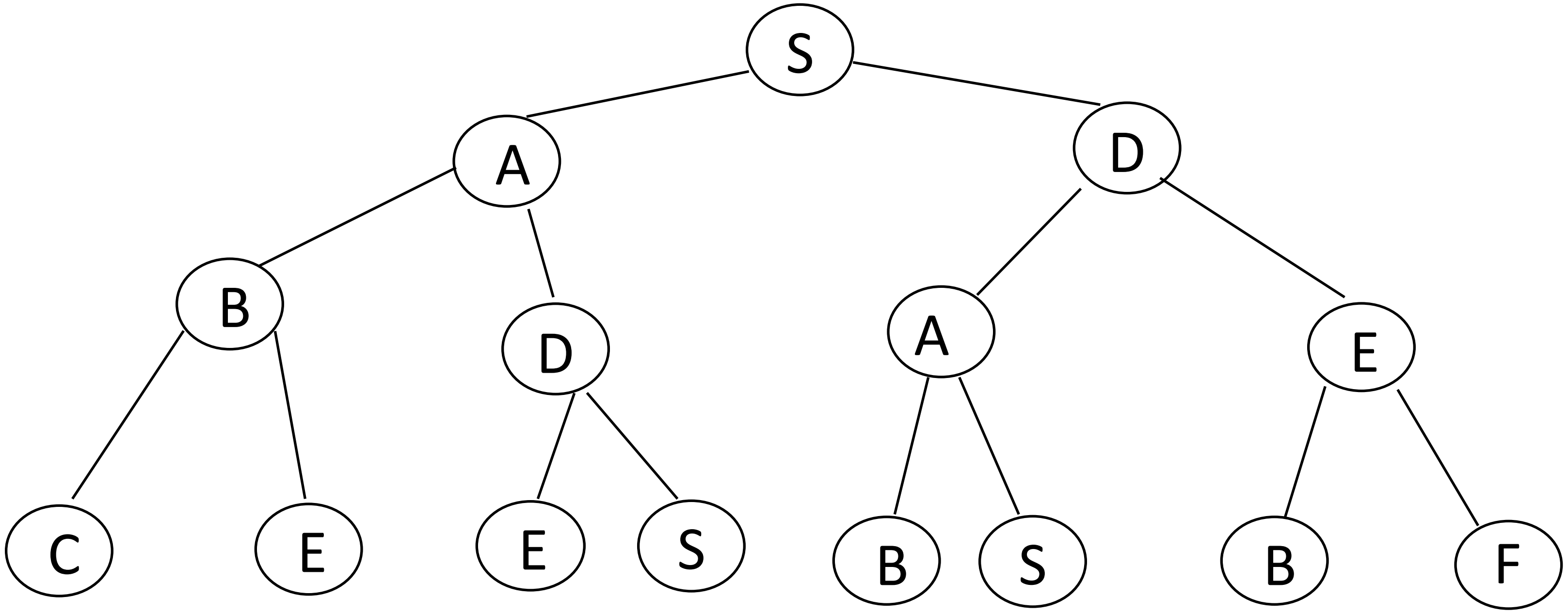


Here, to avoid repeated states assume we don't expand any child node which appears already in the path from the root S to the parent. (Other strategies are also possible)

Pseudocode for Depth-First Search

```
Initialize: Let Q = {S}
While Q is not empty
        pull Q1, the first element in Q
        if Q1 is a goal
                report(success) and quit
        else
                child_nodes = expand(Q1)
                eliminate child_nodes which represent loops
                put remaining child_nodes at the front of Q
        end
Continue
```

Breadth First Search



(Use the simple heuristic of not generating a child node if that node is a parent to avoid “obvious” loops: this clearly does not avoid all loops and there are other ways to do this)

Comparing DFS and BFS

- Same Time Complexity, unless...
  - say we have a search problem with
    - goals at some depth d
    - but paths without goals and which have infinite depth (i.e., loops in the search space)
  - in this case DFS never may never find a goal!
      - (it stays on an infinite (non-goal) path forever)
  - BFS does not have this problem
      - it will find the finite depth goals in time $O(b^d)$

- Practical considerations
  - if there are no infinite paths, and many possible goals in the search tree, DFS will work best
  - For large branching factors b, BFS may run out of memory
  - BFS is "safer" if we know there can be loops

Depth-Limited Search

- This is Depth-first Search with a cutoff on the maximum depth of any path
  - i.e., implement the usual DFS algorithm
  - when any path gets to be of length m, then do not expand this path any further and backup
  - this will systematically explore a search tree of depth m
- Properties of DLS
  - Time complexity = O(b^m), Space complexity = O(bm)
  - If goal state is within m steps from S:
    - DLS is complete
    - e.g., with N cities, we know that if there is a path to goal state G it can be of length N-1 at most
  - But usually we don't know where the goal is!
    - if goal state is more than m steps from S, DLS is incomplete!
    - => the big problem is how to choose the value of m

Iterative Deepening Search

- Basic Idea:
  - we can run DFS with a maximum depth constraint, m
    - i.e., DFS algorithm but it **backs-up at depth m**
    - this avoids the problem of infinite paths
  - But how do we choose m in practice? say m < d (!!)
  - We can run DFS multiple times, gradually increasing m
    - this is known as Iterative Deepening Search

Procedure

```
for m = 1 to infinity
        if (depth-first search with max-depth = m ) returns success
                then report (success) and quit
        else
                continue
end
```

Iterative Deepening Search

- Complexity
  - Space complexity = $O(bd)$
    - (since its like depth first search run different times)
  - Time Complexity
    - $1 + (1+b) + (1 + b+b^2) + ...(1 + b+....b^d)$
      $= O(b^d)$
      (i.e., the same as BFS or DFS in the the worst case)

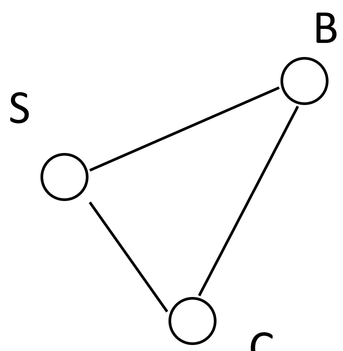
    - The overhead in repeated searching of the same subtrees is small relative to the overall time
      - e.g., for b=10, only takes about 11% more time than DFS

- A useful practical method
  - combines
    - guarantee of finding a solution if one exists (as in BFS)
    - space efficiency, $O(bd)$ of DFS

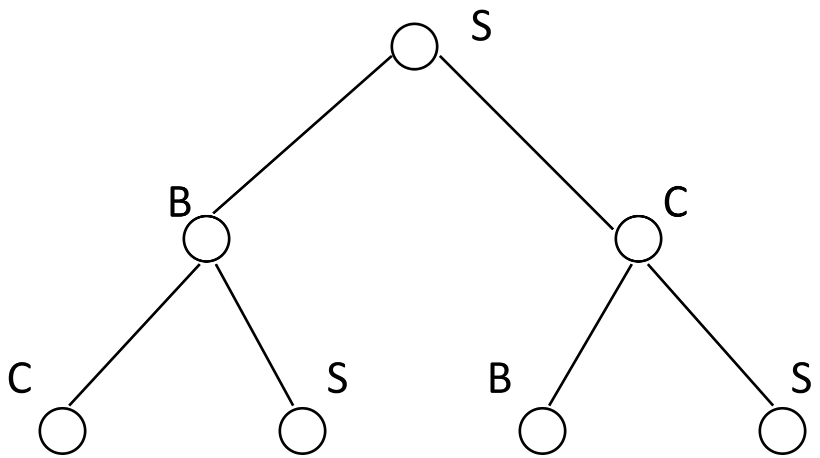
Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both “meet in the middle”
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?
- Complexity
  - time complexity is $O(2\ b^{(d/2)}) = O(b^{(d/2)})$ steps
  - memory complexity is the same

Repeated States



State Space



Example of a Search Tree

- For many problems we can have repeated states in the search tree
  - i.e., the same state can be gotten to by different paths
  - => same state appears in multiple places in the tree
    - this is inefficient, we want to avoid it

- How inefficient can this be?
  - a problem with a finite number of states can have an infinite search tree!

Techniques for Avoiding Repeated States

- Method 1
  - when expanding, do not allow return to parent state
  - (but this will not avoid “triangle loops” for example)

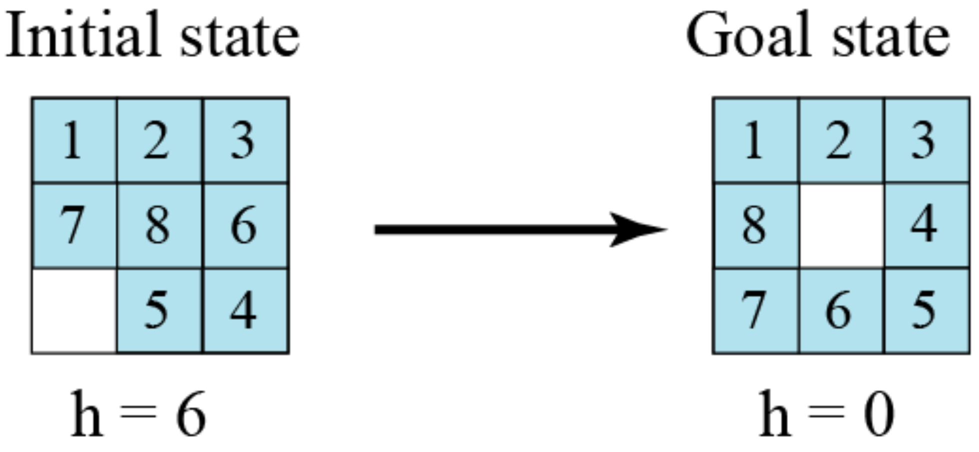
- Method 2
  - do not create paths containing cycles (loops)
  - i.e., do not keep any child-node which is also an ancestor in the tree

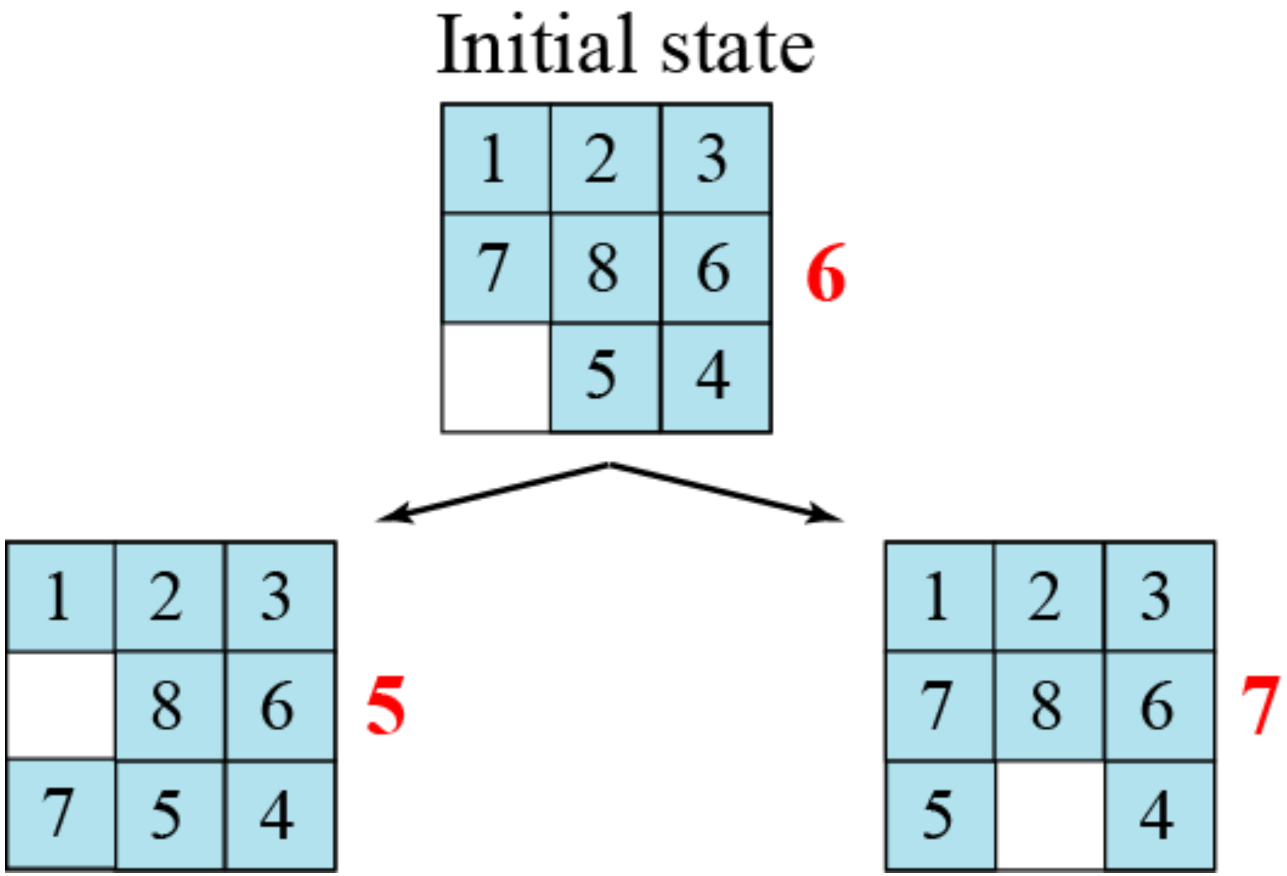
- Method 3
  - never generate a state generated before
    - only method which is guaranteed to always avoid repeated states
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states

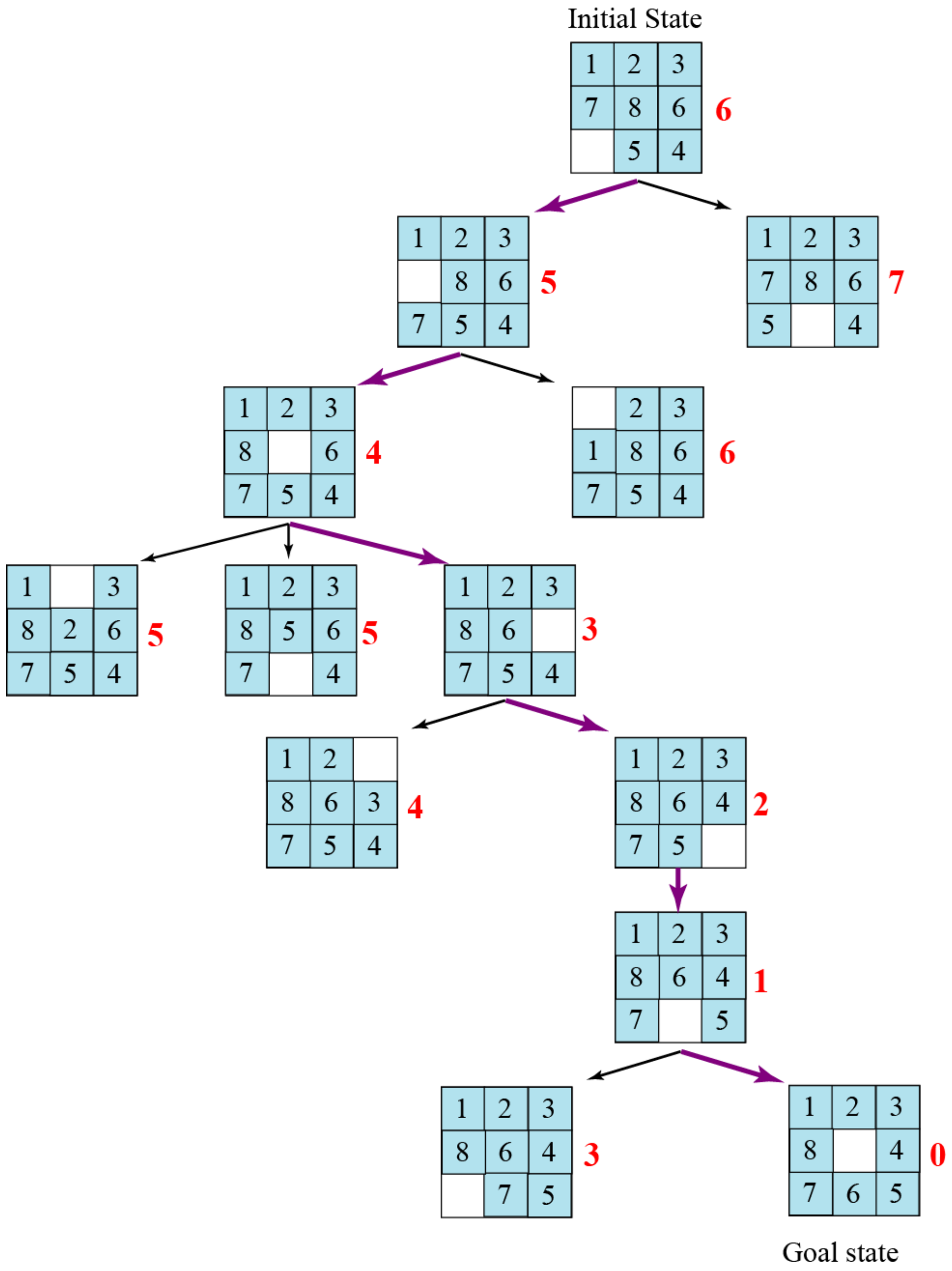
- Methods 1 and 2 are most practical, work well on most problems

Heuristic search

Using heuristic search, we assign a quantitative value called a heuristic value (h value) to each node. This quantitative value shows the relative closeness of the node to the goal state. For example, consider solving the 8-puzzle.




Initial state
1
2
3
7
8
6
5
4
6
1
2
3
8
6
7
5
4
5
1
2
3
7
8
6
5
4
7


Initial State
Goal state

Uniform Cost Search

- **Uniform Cost Search**
  - orders the nodes on the Q according to path cost from S
  - always expands the node with minimum path cost from S

```
Initialize: Let Q = {S}
While Q is not empty
        pull Q1, the first element in Q
        if Q1 is a goal report(success) and quit
        else
                child_nodes = expand(Q1)
                <eliminate child_nodes which represent loops>
                put remaining child_nodes in Q
                sort Q according to path-cost to each node
        end
Continue
```

Heuristics and Search

- in general
  - a heuristic is a “rule-of-thumb” based on domain-dependent knowledge to help you solve a problem

- in search
  - one uses a heuristic function of a state where

    h(node) = estimated cost of cheapest path from the state for that node to a goal state G

    - h(G) = 0
    - h(other nodes) $\geq$ 0
    - (note: we will assume all individual node-to-node costs are > 0)

A(*) Algorithm

- Goal: Find shortest path
- Prerequisites
  - Graph
  - Method to estimate distance between points (heuristic)
- Basic Method
  - Try all paths?
    - Takes time
  - Orient search towards target
    - Minimizes areas of the map to be examined
    - Uses heuristics that indicate the estimated cost of getting to the destination
    - Main advantage

A(*) Algorithm

- Algorithm
  - Open list
    - Nodes that need to be considered as possible starts for further extensions of the path
  - Closed list
    - Nodes that have had all their neighbors added to the open list
  - G score
    - Contains the length or weight of the path from the current node to the start node
    - Low lengths are better
    - Every node has a G score
  - H score
    - Heuristic
    - Resembles G score except it represents an estimate of the distance from the current node to the endpoint
    - To find shortest path, this score must underestimate the distance

The A* Algorithm

- A heuristic h is admissible if
  - it for any node n it does NOT overestimate the true path cost from n to the nearest goal.

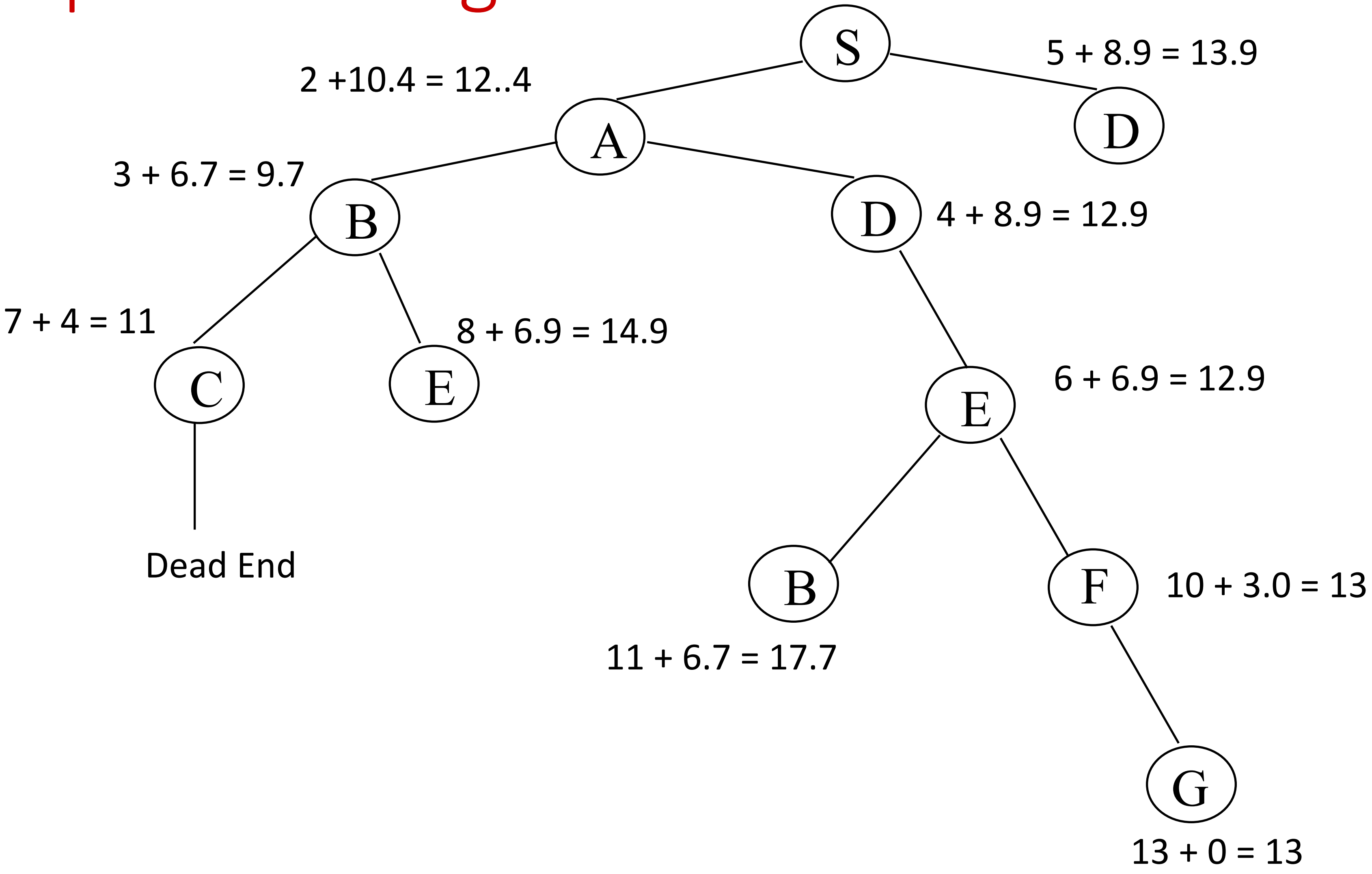
- The A* search is a search algorithm orders the nodes on the Q according to f(n)=g(n)+h(n), where h(n) is an admissible heuristic
  - i.e., it sorts nodes on Q according to an admissible heuristic h*
  - It is like uniform-cost,
    - but uses fcost(node) = path-cost(S to node) + h(node)
    - rather than just path cost(S to node)

  - note that uniform cost search can be viewed as A* search where h(n) equals 0 for all n (the latter heuristic equal to 0 for every node is clearly admissible! Why?)

Pseudo-code for the A* Search Algorithm

```
Initialize: Let Q = {S}
While Q is not empty
      pull Q1, the first element in Q
      if Q1 is a goal report(success) and quit
      else
                child_nodes = expand(Q1)
                <eliminate child_nodes which represent loops>
                put remaining child_nodes in Q
                sort Q according to ucost = pathcost(S to node) + h*(node)
      end
Continue
```

Example of A* Algorithm in action



Comments on heuristic estimation

- The estimate of the distance is called a heuristic
  - typically it comes from domain knowledge
  - e.g., the straight-line distance between 2 points
- If the heuristic never overestimates, then the search procedure using this heuristic is "admissible", i.e.,
  - h*(N) is less than or equal to realcost(N to G)
- A* is a search with admissible heuristic is optimal
  - i.e., if one uses an admissible heuristic to order the search one is guaranteed to find the optimal solution
- The closer the heuristic is to the real (unknown) path cost, the more effective it will be, ie if h1(n) and h2(n) are two admissible heuristics and $h1(n) \leq h2(n)$ for any node n then A* search with h2(n) will in general expand fewer nodes than A* search with h1(n)

Properties of A*

- A* generates an optimal solution if h(n) is an admissible heuristic and the search space is a tree:
  - h(n) is **admissible** if it never overestimates the cost to reach the destination node

- A* generates an optimal solution if h(n) is a consistent heuristic and the search space is a graph:
  - h(n) is **consistent** if for every node n and for every successor node n' of n:

$$h(n) \leq c(n,n') + h(n')$$

- If h(n) is consistent then h(n) is admissible
- Frequently when h(n) is admissible, it is also consistent

Admissible Heuristics

- A heuristic is admissible if it is too optimistic, estimating the cost to be smaller than it actually is.

- Example:

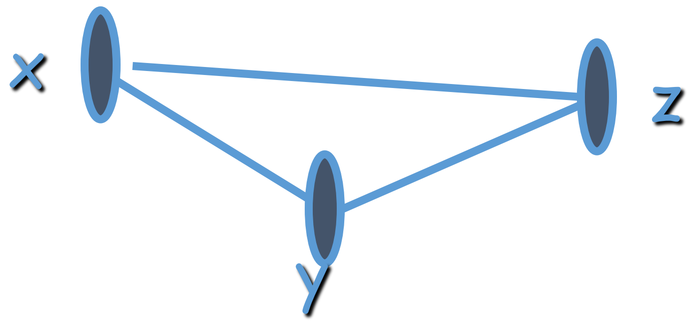
  In the road map domain,

  h(n) = "Euclidean distance to destination"

  is admissible as normally cities are not connected by roads that make straight lines

Metric Space

- A set of points X
- Distance function d(x,y)
  $d : X \rightarrow [0 \ldots \infty)$
  - $d(x,y) = 0$ iff $x=y$
  - $d(x,y) = d(y,x)$ Symmetric
  - $d(x,z) \leq d(x,y) + d(y,z)$ Triangle inequality



- Metric space M(X,d)

Dominance

If $h2(n) \geq h1(n)$ for all n (both admissible)

then h2 dominates h1

h2 is better for search: it is guaranteed to expand
less or equal nr of nodes.

Examples of Heuristic Functions for A*

- the 8-puzzle problem
  - the number of tiles in the wrong position
    - is this admissible?
  - the sum of distances of the tiles from their goal positions, where distance is counted as the sum of vertical and horizontal tile displacements (“Manhattan distance”)
    - is this admissible?

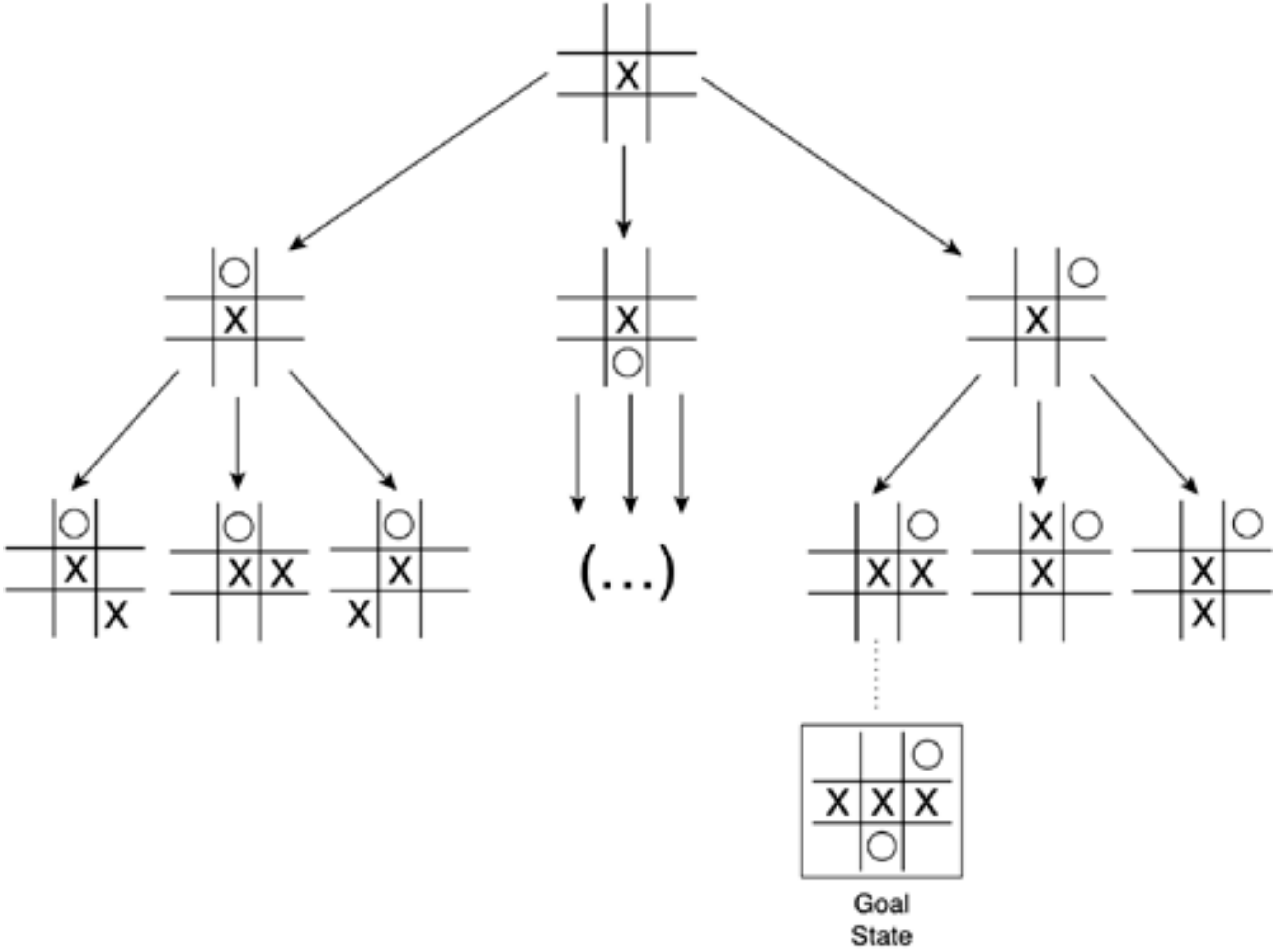
- How can we invent admissible heuristics in general?
  - look at “relaxed” problem where constraints are removed
    - e.g., we can move in straight lines between cities
    - e.g., we can move tiles independently of each other

IDA(*) Algorithm

- A*, like depth-first search, except based on increasing values of total cost rather than increasing depths
- IDA* sets bounds on the heuristic cost of a path, instead of depth
- A* always finds a cheapest solution if the heuristic is admissible
- IDA* is optimal in terms of solution cost, time, and space for admissible best-first searches on a tree

State Space

Model of a system as a set of input, output and state variables



Setting Up a State Space Model

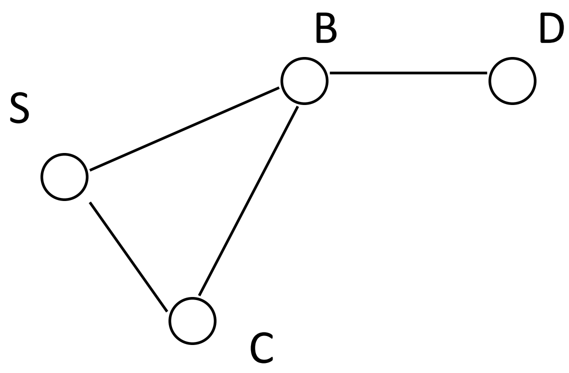
- State-space Model is a Model for The Search Problem
  - usually a set of discrete states X
    - e.g., in driving, the states in the model could be towns/cities

- Start State - a state from X where the search starts

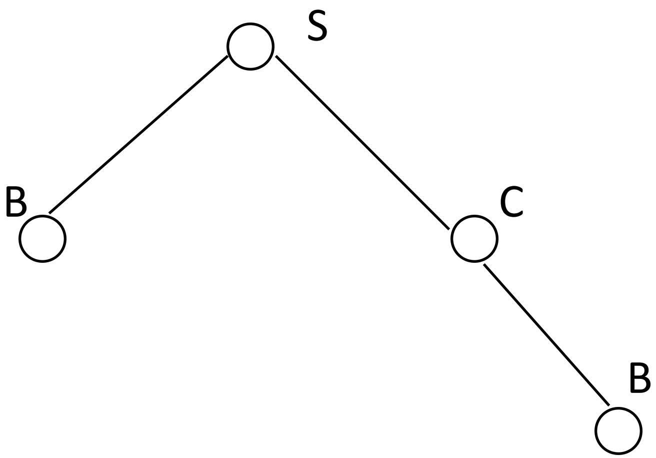
- Goal State(s)
  - a goal is defined as a target state
  - For now: all goal states have utility 1, and all non-goals have utility 0
  - there may be many states which satisfy the goal
    - e.g., drive to a town with an airport
  - or just one state which satisfies the goal
    - e.g., drive to Las Vegas

- Operators
  - operators are mappings from X to X
    - e.g. moves from one city to another that are legal (connected with a road)

A State Space and a Search Tree are different



State Space



Example of a Search Tree

- A State Space represents all states and operators for the problem
- A Search Tree is what an algorithm constructs as it solves a search problem:
  - so we can have different search trees for the same problem
  - search trees grow in a dynamic fashion until the goal is found

Puzzle-Solving as Search

- You have a 3-gallon and a 4-gallon
- You have a faucet with an unlimited amount of water
- You need to get exactly 2 gallons in 4-gallon jug

- State representation: (x, y)
  - x: Contents of four gallon
  - y: Contents of three gallon

- Start state: (0, 0)
- Goal state(s) G = {(2, 0), (2, 1), (2, 2)}
- Operators
  - Fill 3-gallon (0,0)->(0,3), fill 4-gallon (0,0)->(0,4)
  - Fill 3-gallon from 4-gallon (4,0)->(1,3), fill 4-gallon from 3-gallon (0,3)->(3,0) or (1,3)->(4,0) or (2,3)->(4,0)….
  - Empty 3-gallon into 4-gallon, empty 4-gallon into 3-gallon
  - Dump 3-gallon down drain (0,3)->(0,0), dump 4-gallon down drain (4,0)->(0,0)

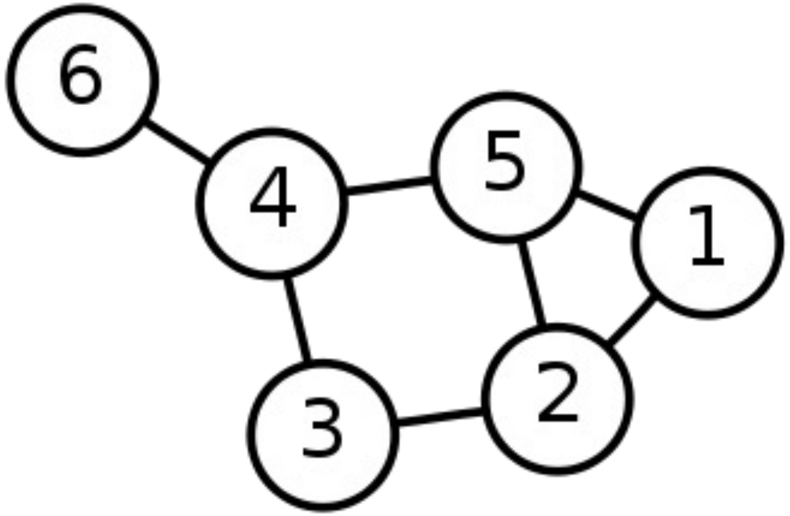
Dijkstra's algorithm

- **Dijkstra's algorithm**: finds shortest (minimum weight) path between a particular pair of vertices in a *weighted* directed graph with nonnegative edge weights
  - solves the "one vertex, shortest path" problem
  - basic algorithm concept: create a table of information about the currently known best way to reach each vertex (distance, previous vertex) and improve it until it reaches the best solution

- in a graph where:
  - vertices represent cities,
  - edge weights represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and any other

Single-Source Shortest Path Problem

<u>Single-Source Shortest Path Problem</u> - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

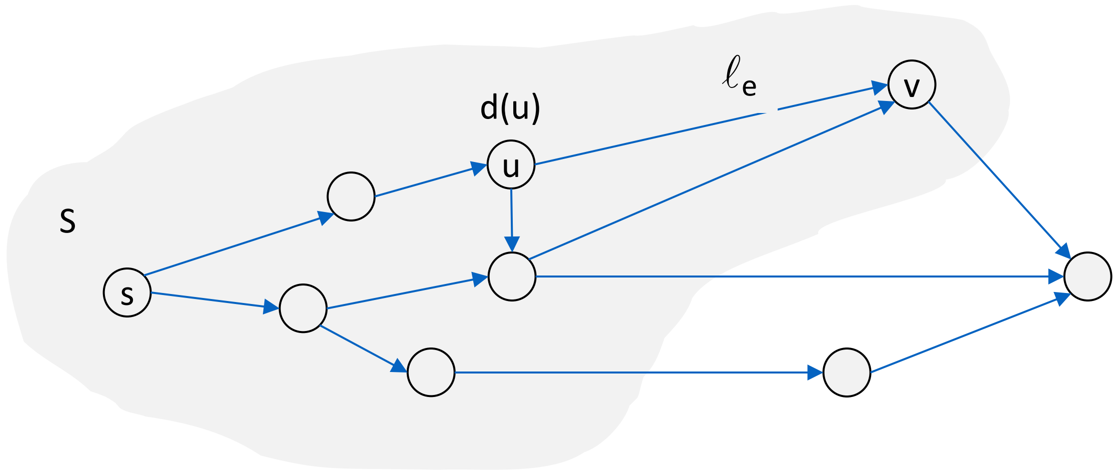
Dijkstra's algorithm - Pseudocode

```
dist[s] ←o                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ←∞                      (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all
vertices)
while Q ≠∅                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)                (select the element of Q with the min.
distance)
      S←S∪{u}                               (add u to list of visited vertices)
       for all v ∈ neighbors[u]
              do  if   dist[v] > dist[u] + w(u, v)             (if new shortest path found)
                         then      d[v] ←d[u] + w(u, v)      (set new value of shortest path)
                                (if desired, add traceback code)
return dist
```

Dijkstra's Algorithm

- Dijkstra's algorithm.
  - Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
  - Initialize S = { s }, d(s) = 0.
  - Repeatedly choose unexplored node v which minimizes

  add v to S, and set d(v) = π(v).

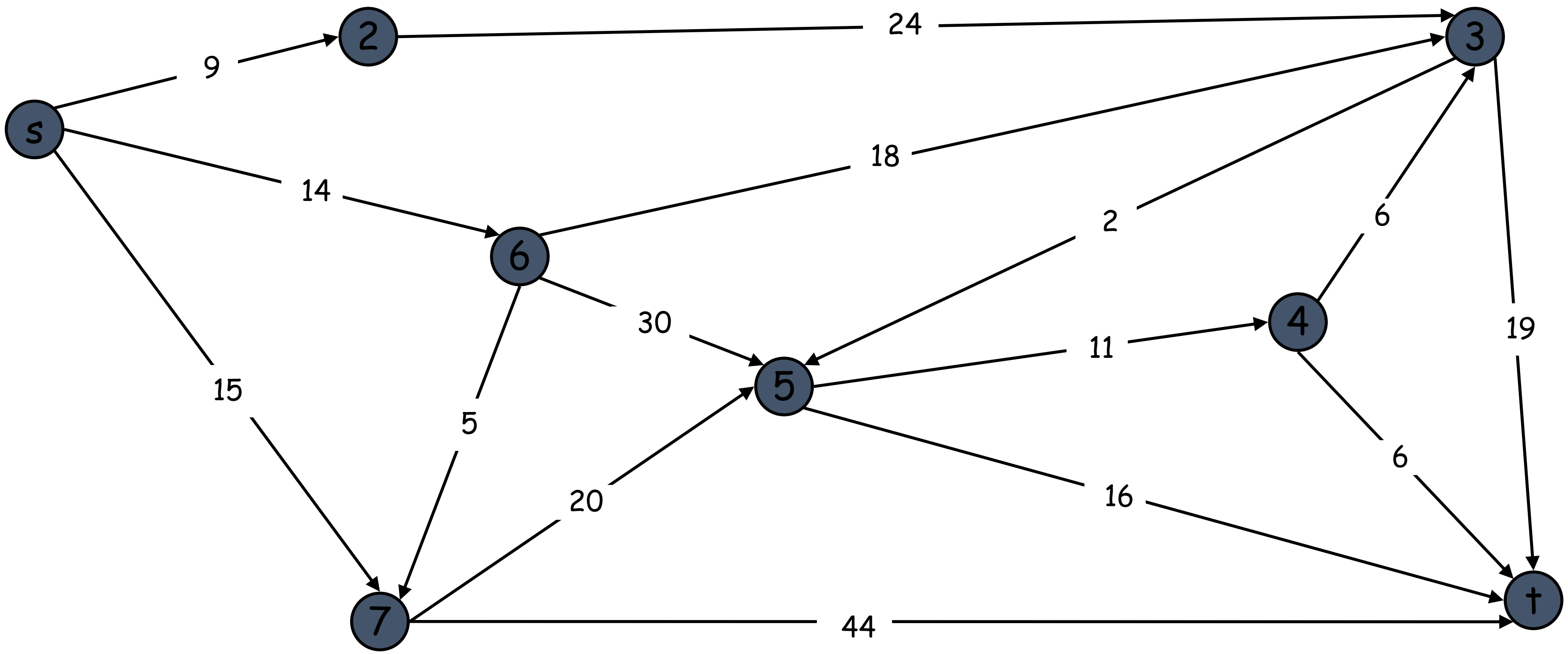


Dijkstra's Algorithm: Proof of Correctness

- Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.
- Pf. (by induction on $|S|$)
- Base case: $|S| = 1$ is trivial.
- Inductive hypothesis: Assume true for $|S| = k \geq 1$.
  - Let v be next node added to S, and let u-v be the chosen edge.
  - The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
  - Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
  - Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
  - P is already too long as soon as it leaves S.

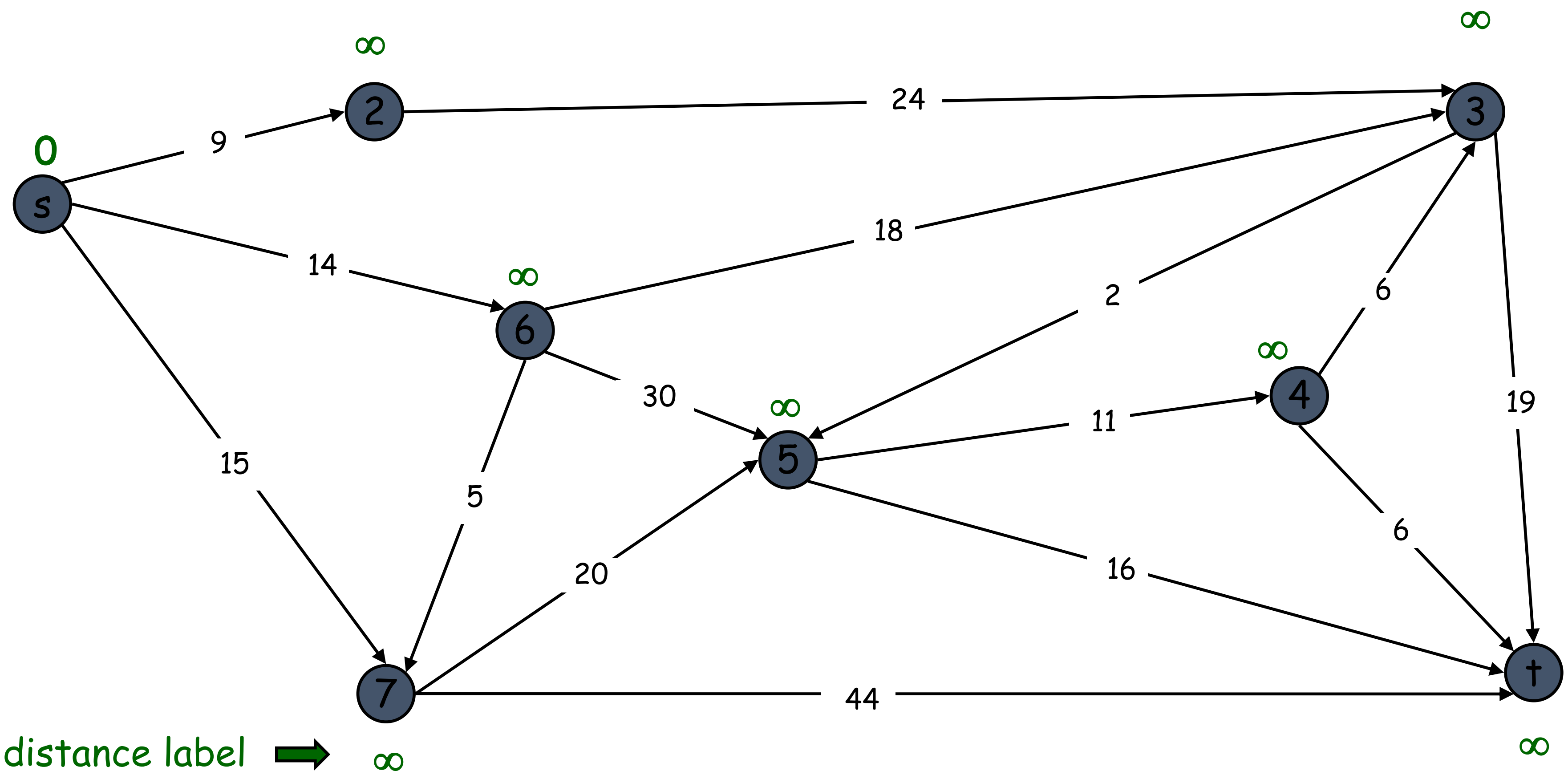
Dijkstra's Shortest Path Algorithm

- Find shortest path from s to t.



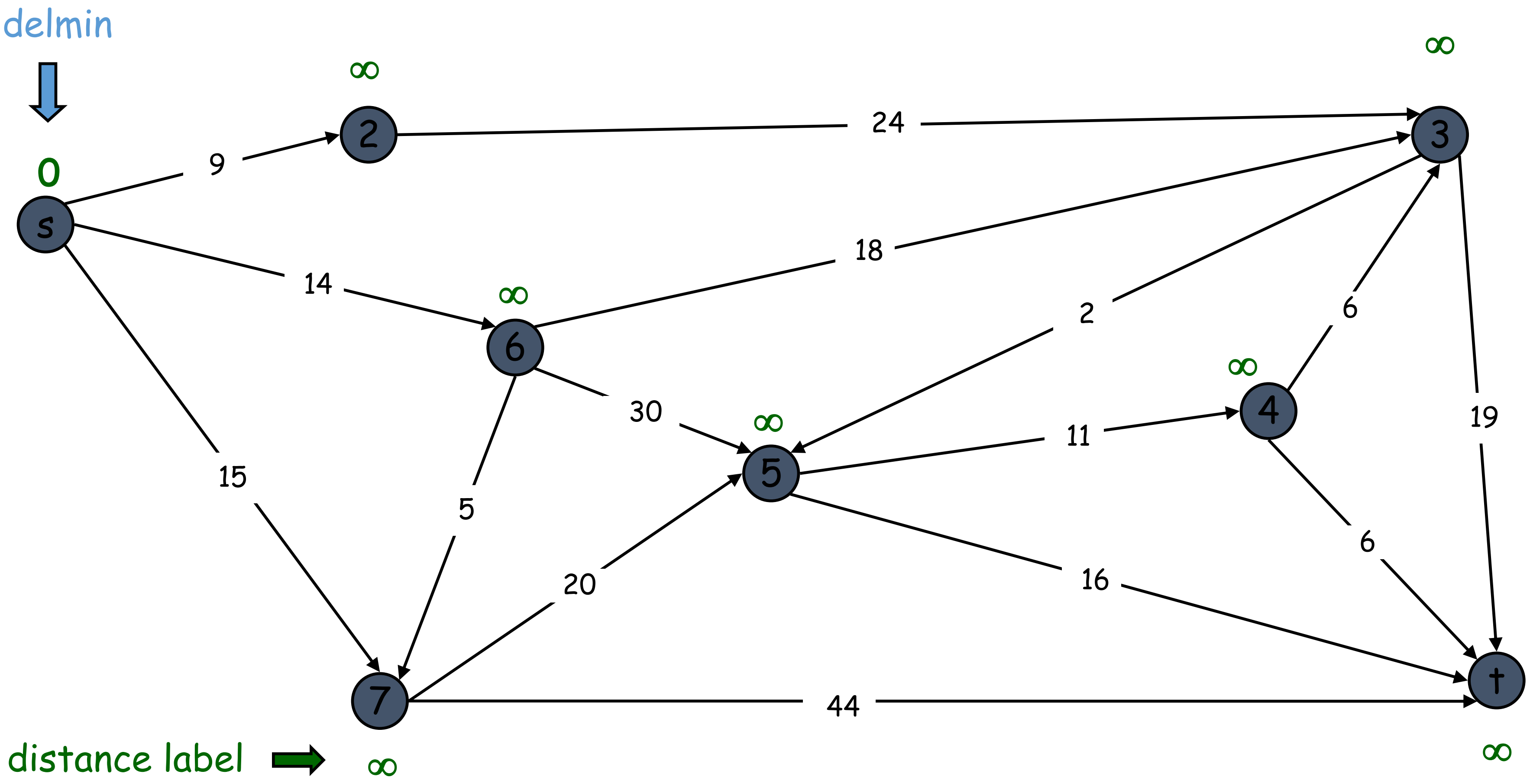
Dijkstra's Shortest Path Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



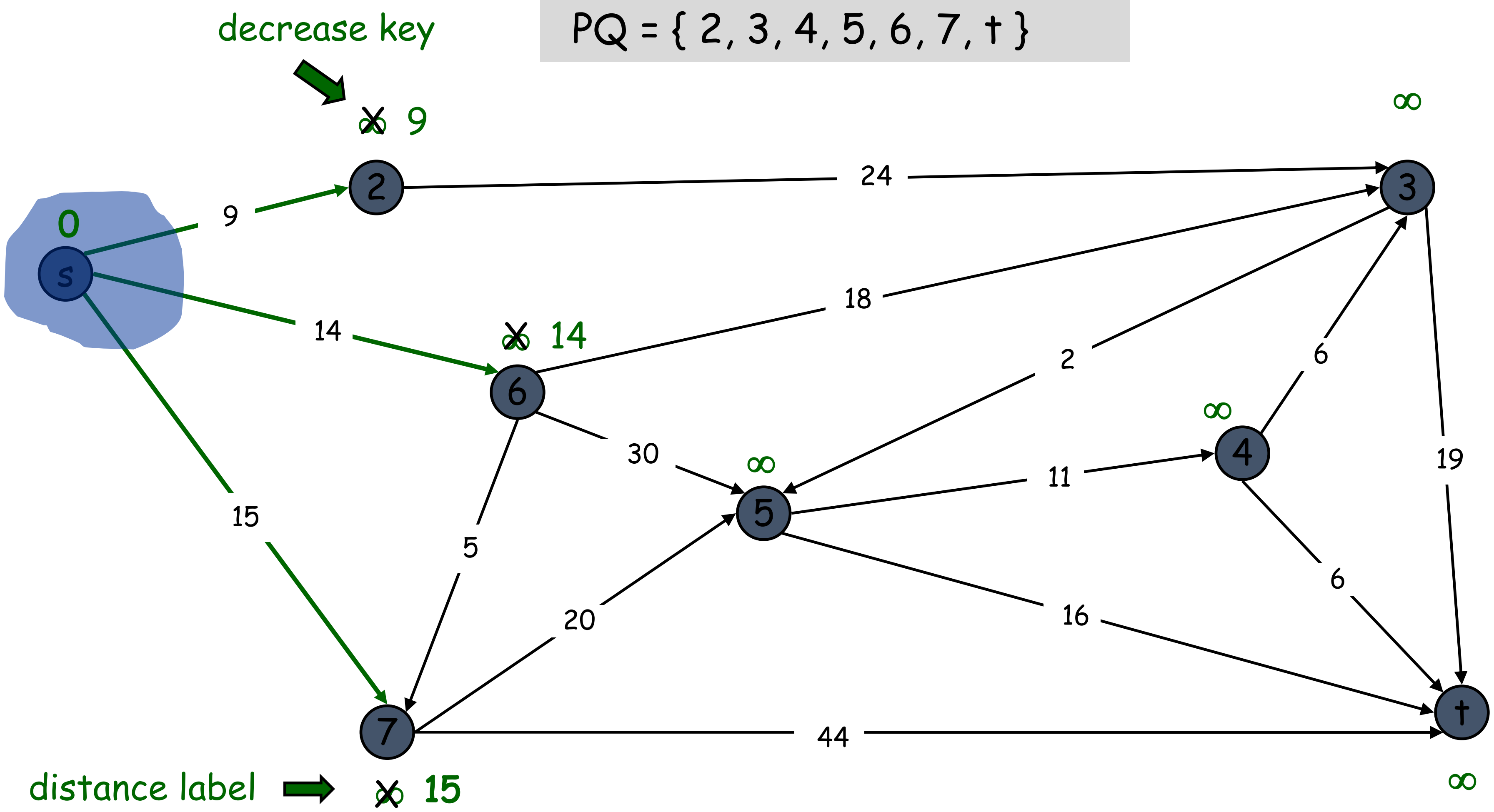
Dijkstra's Shortest Path Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



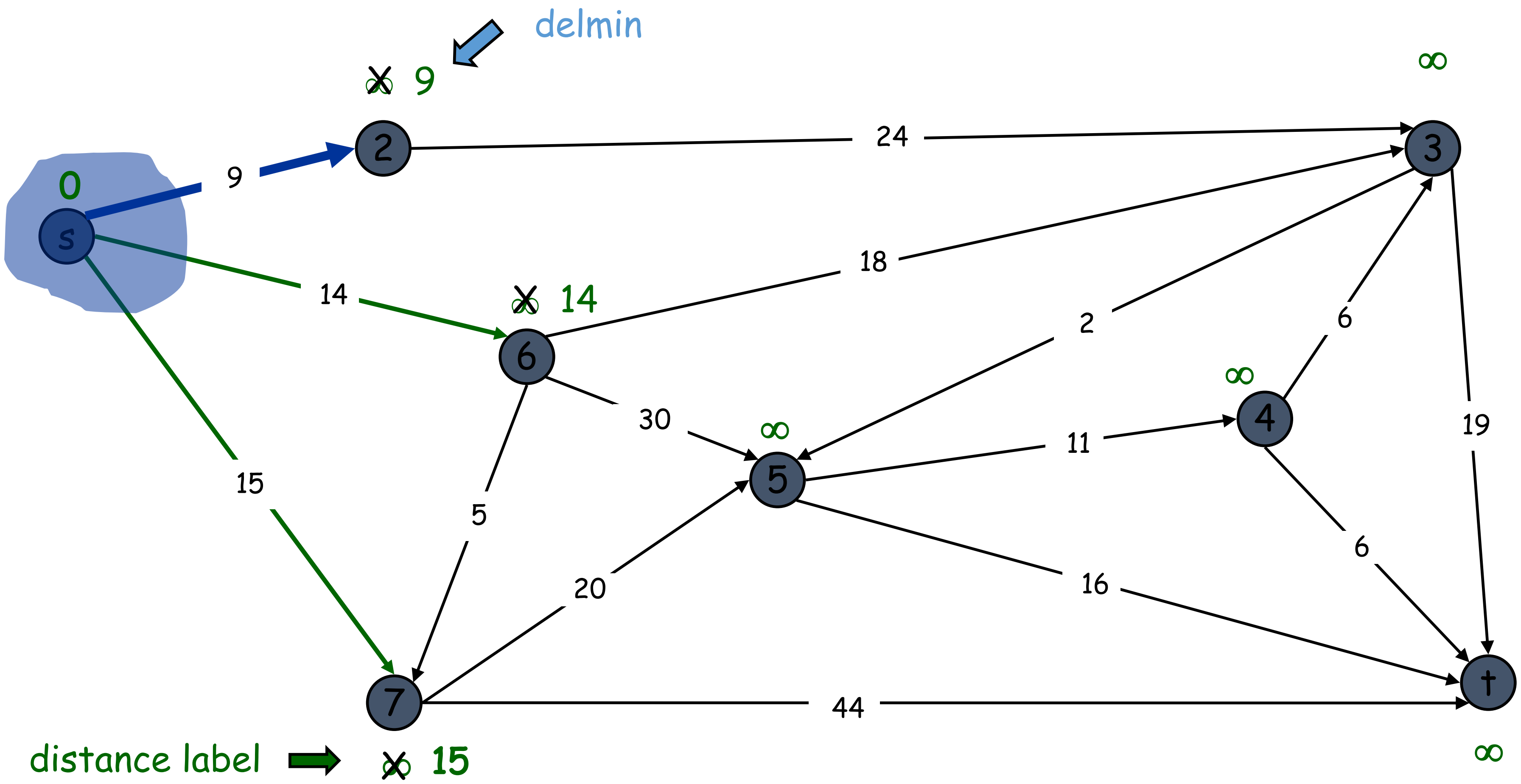
Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



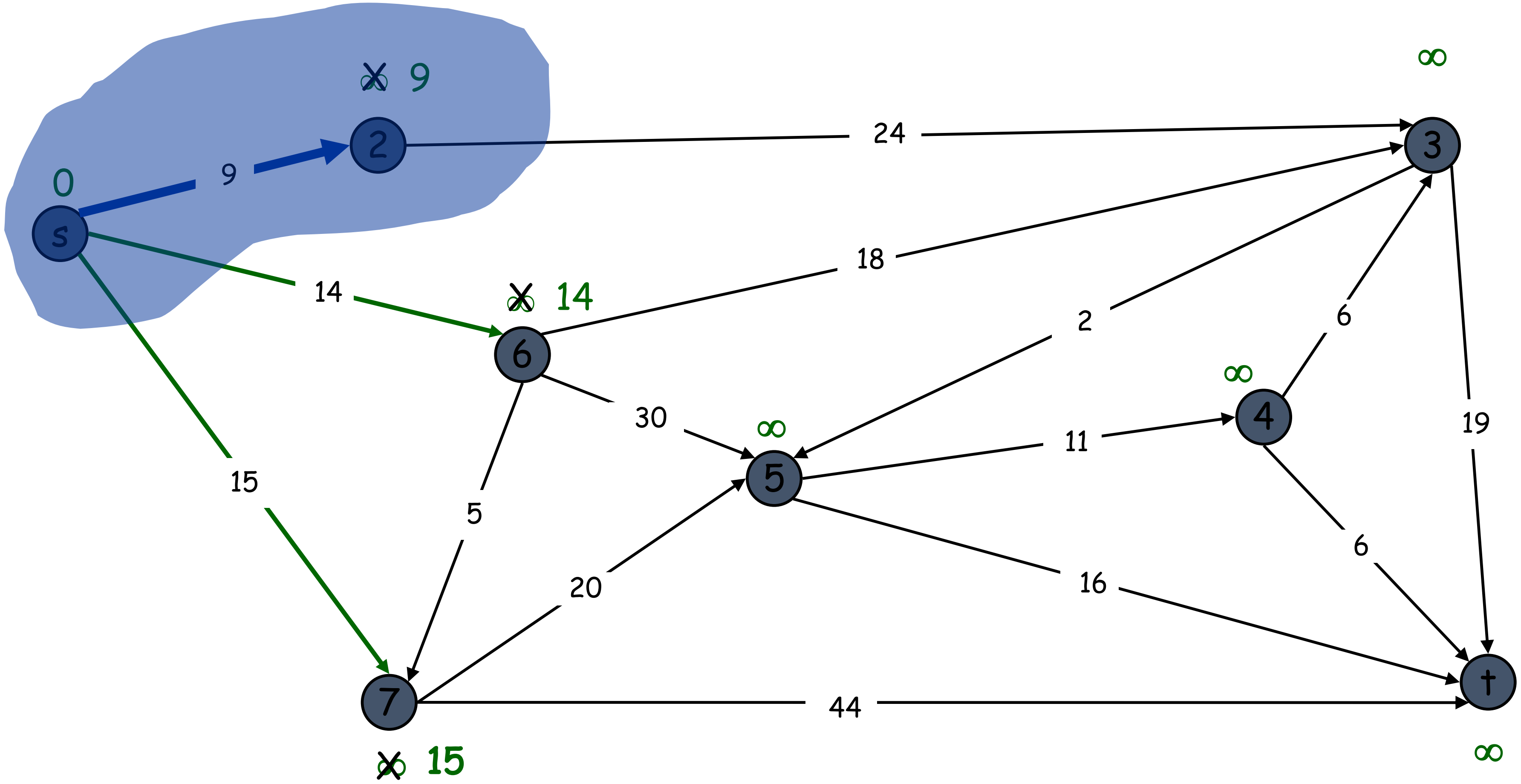
Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



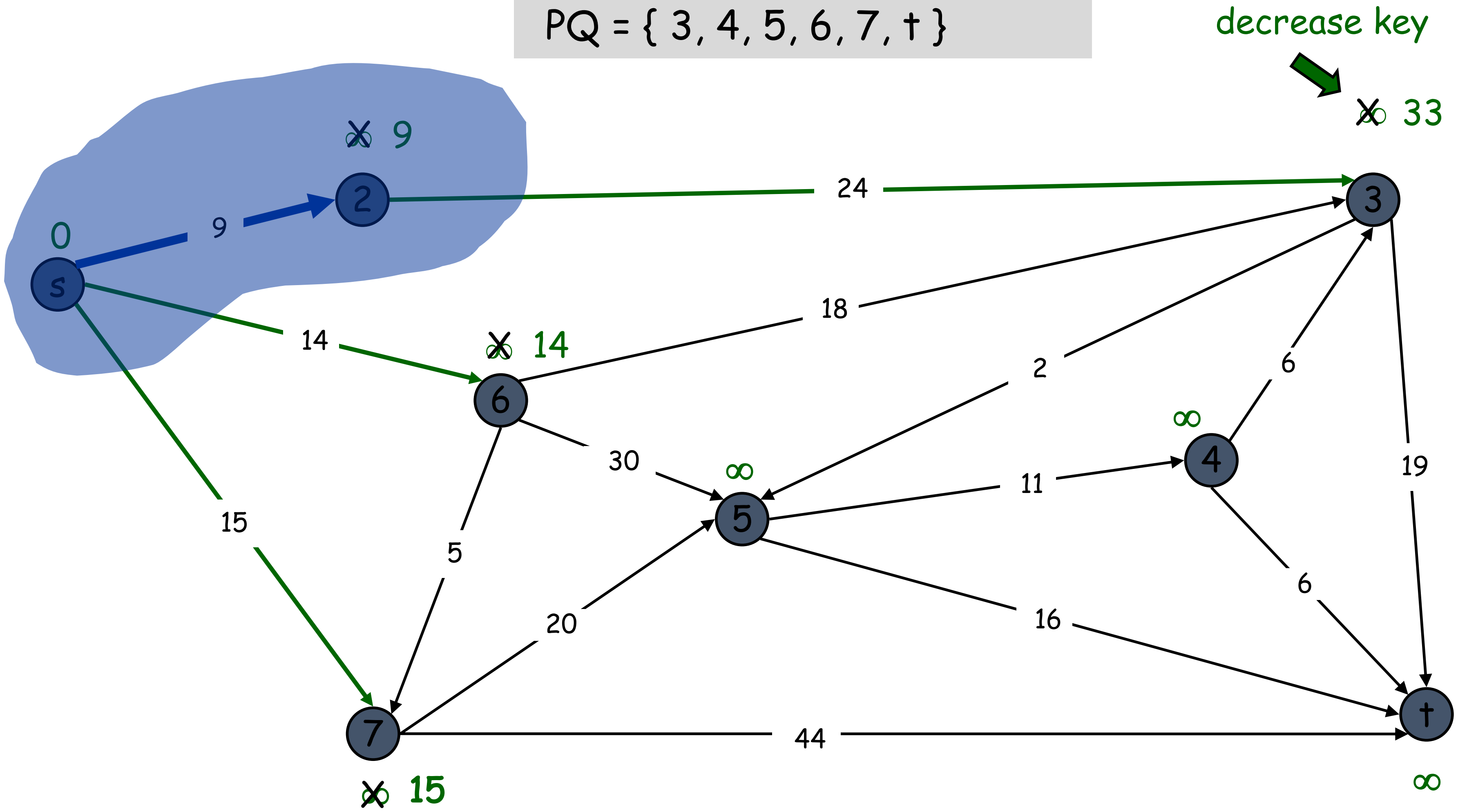
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



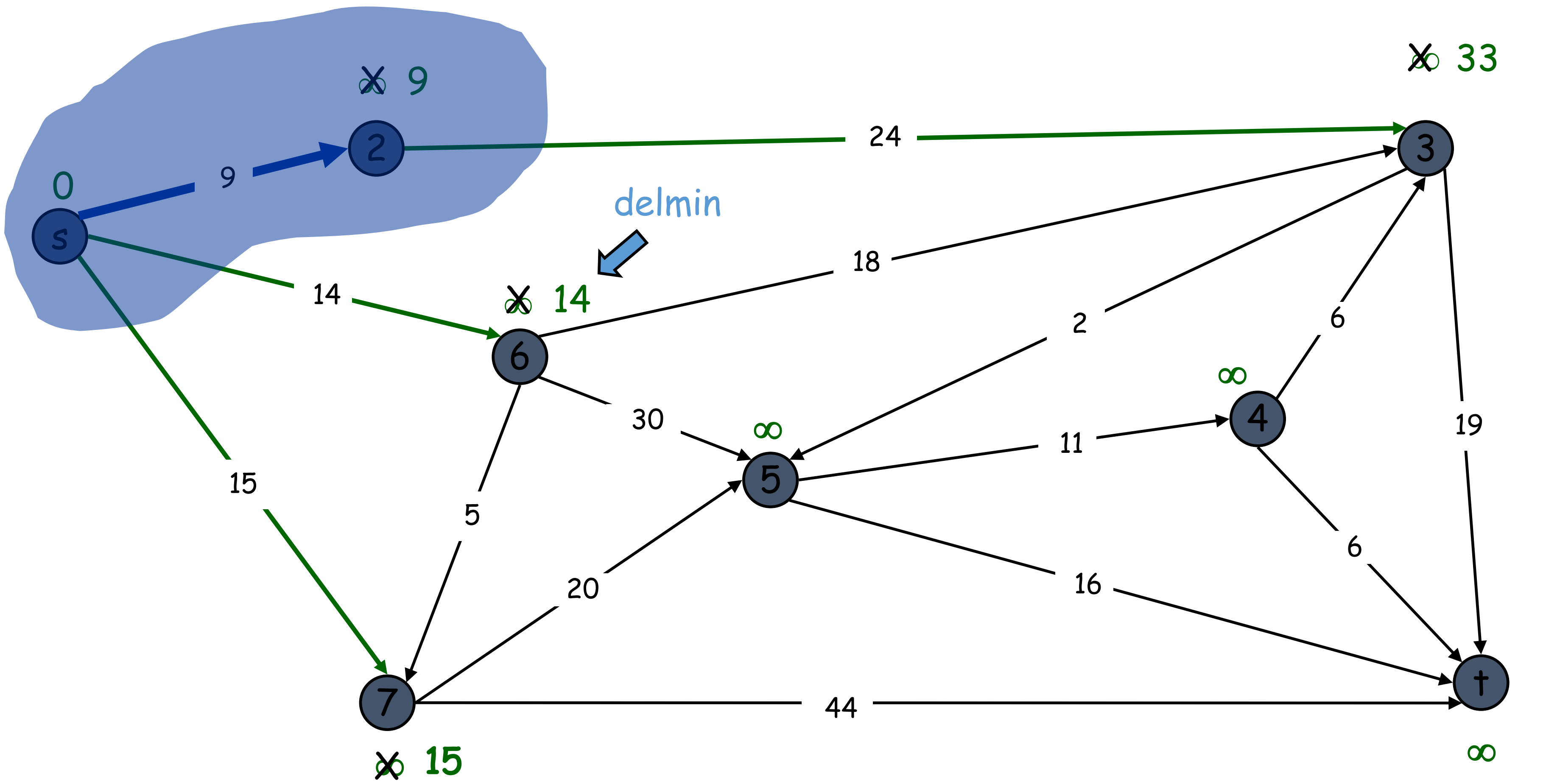
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



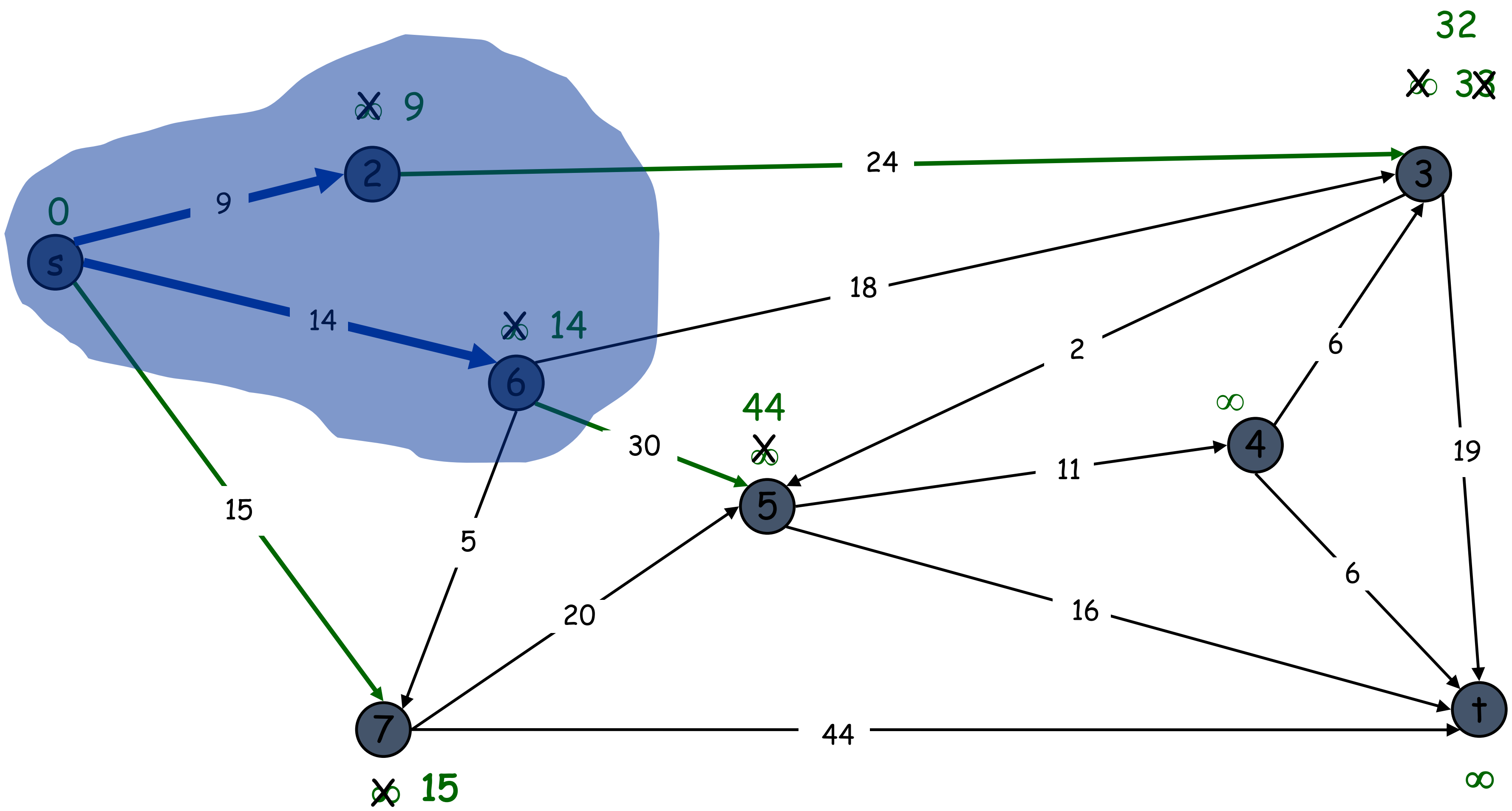
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



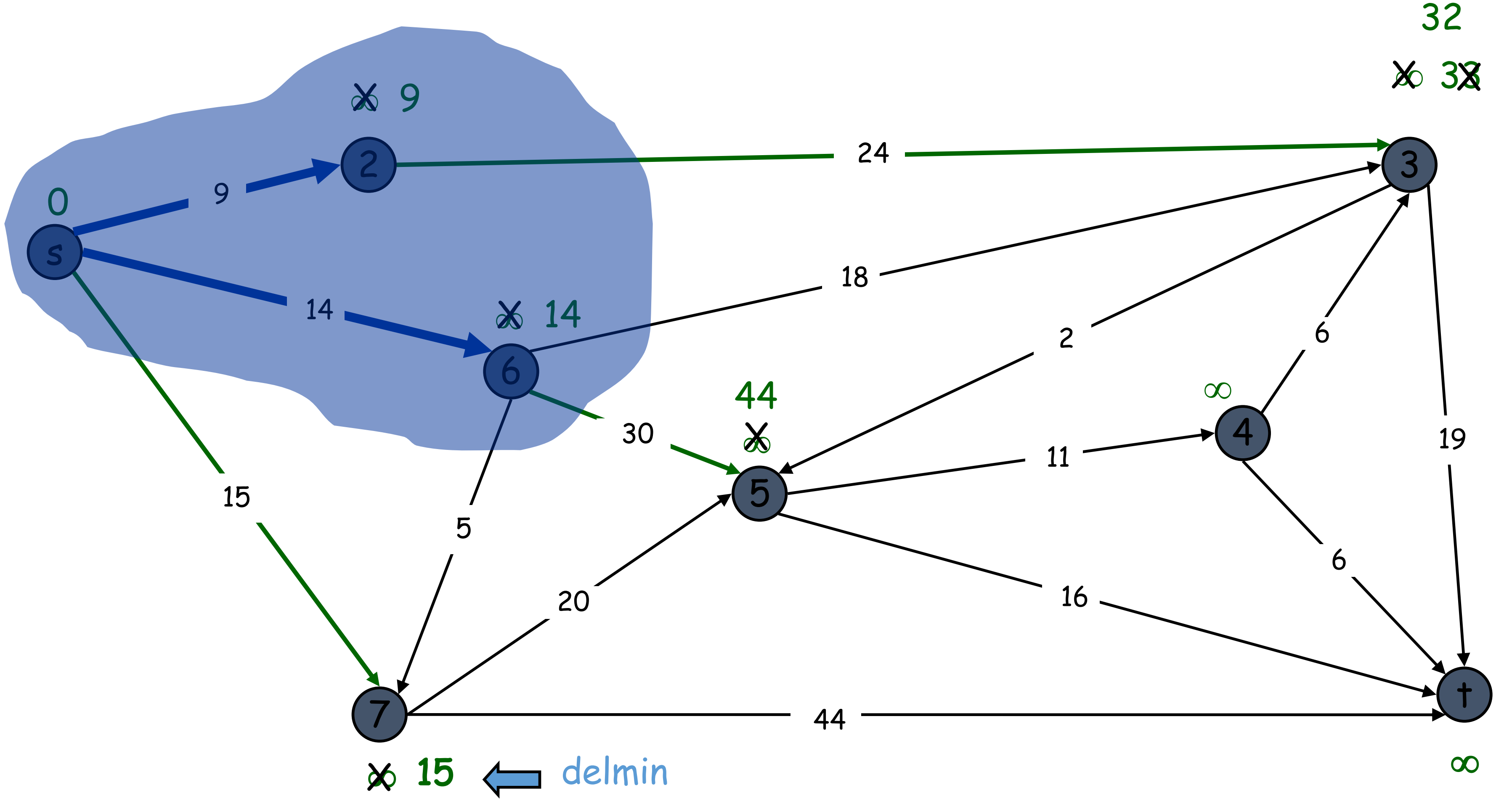
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



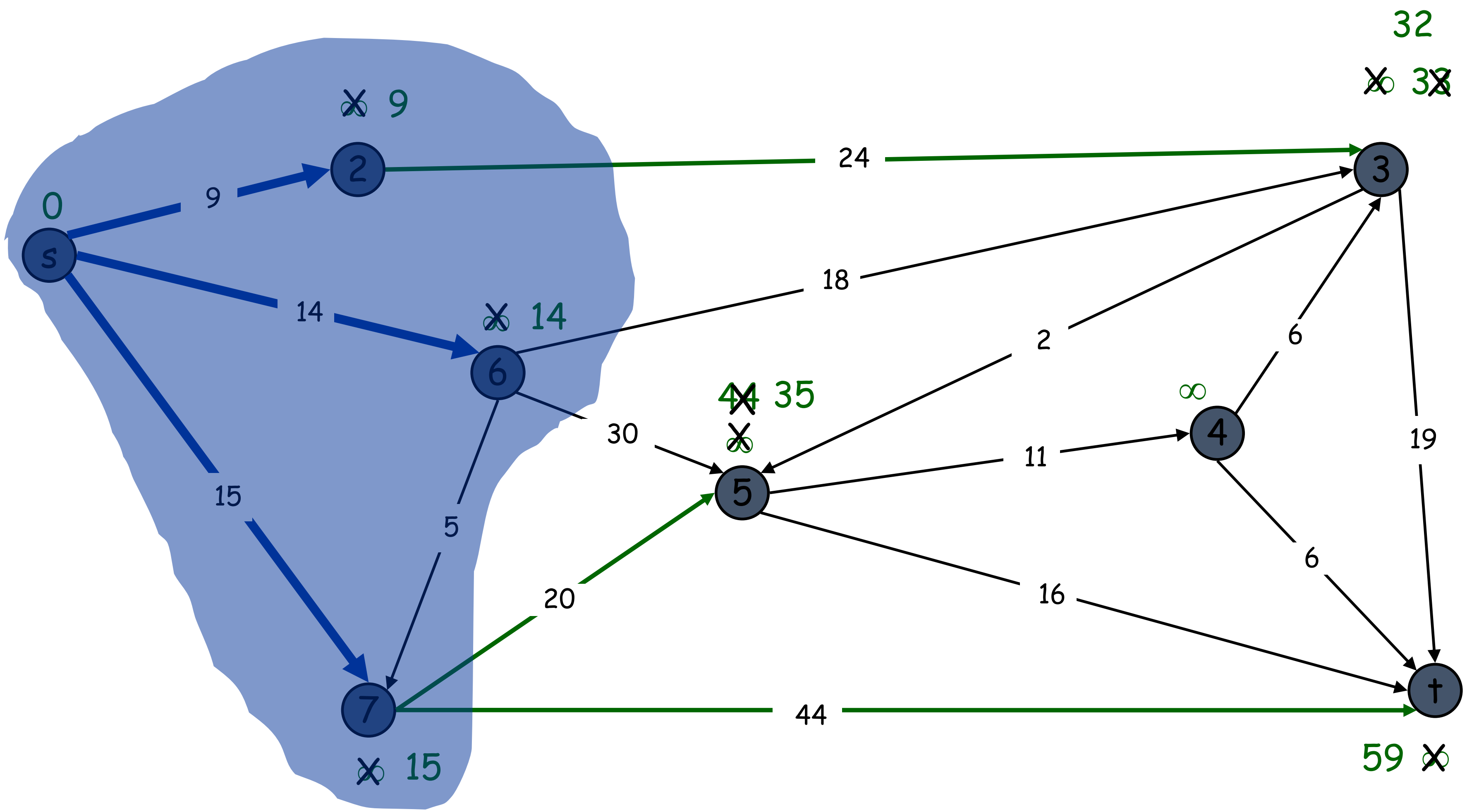
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



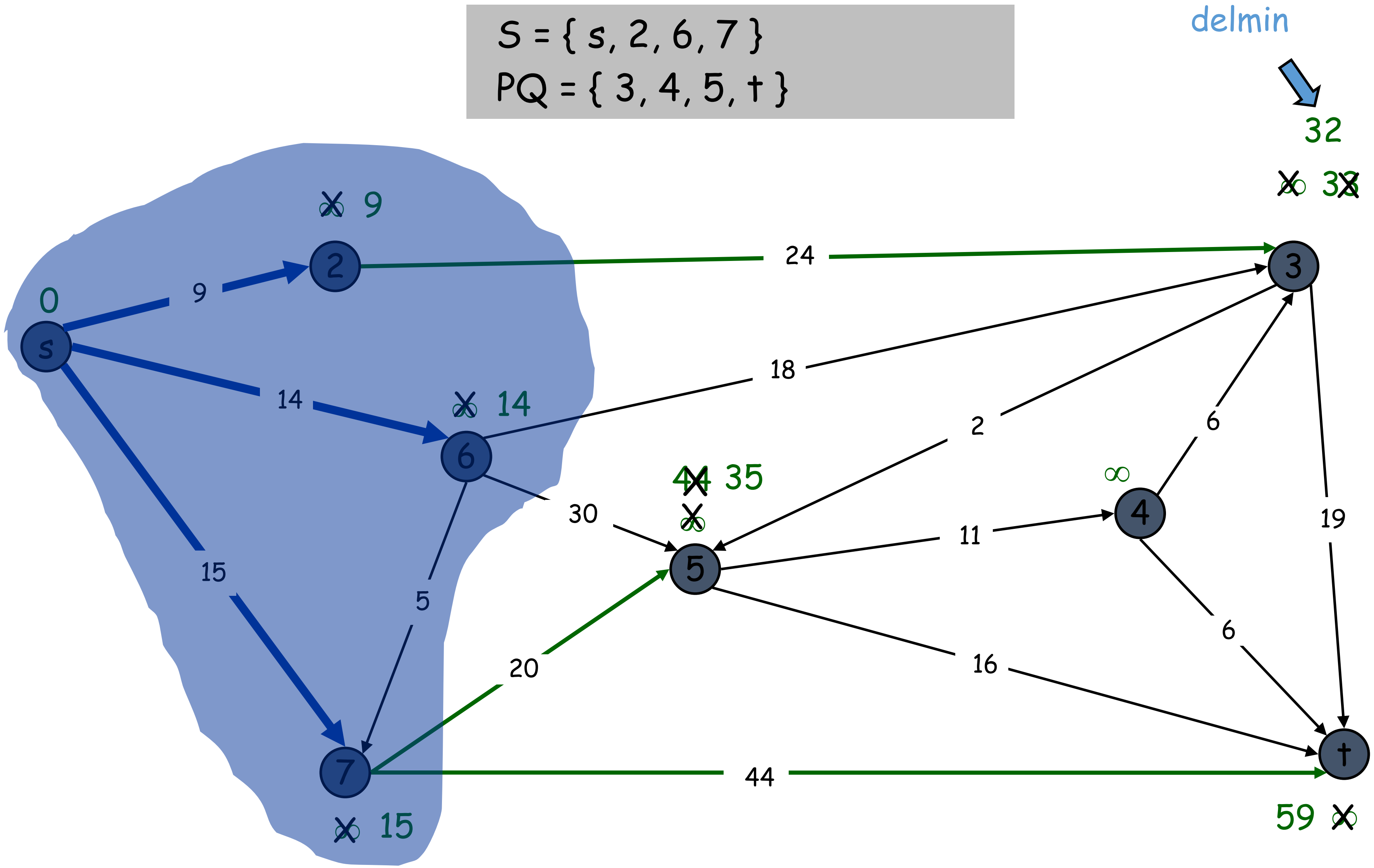
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



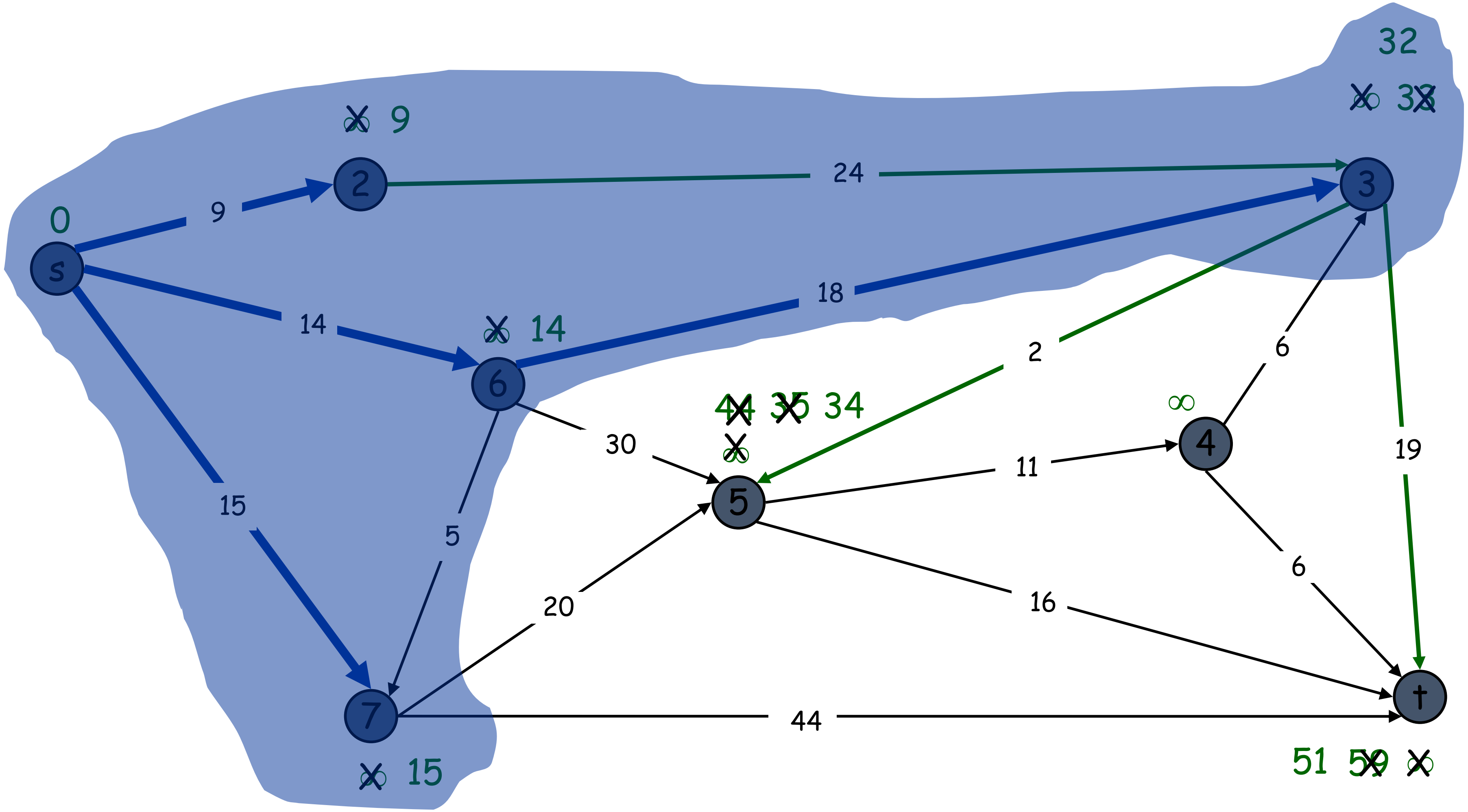
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



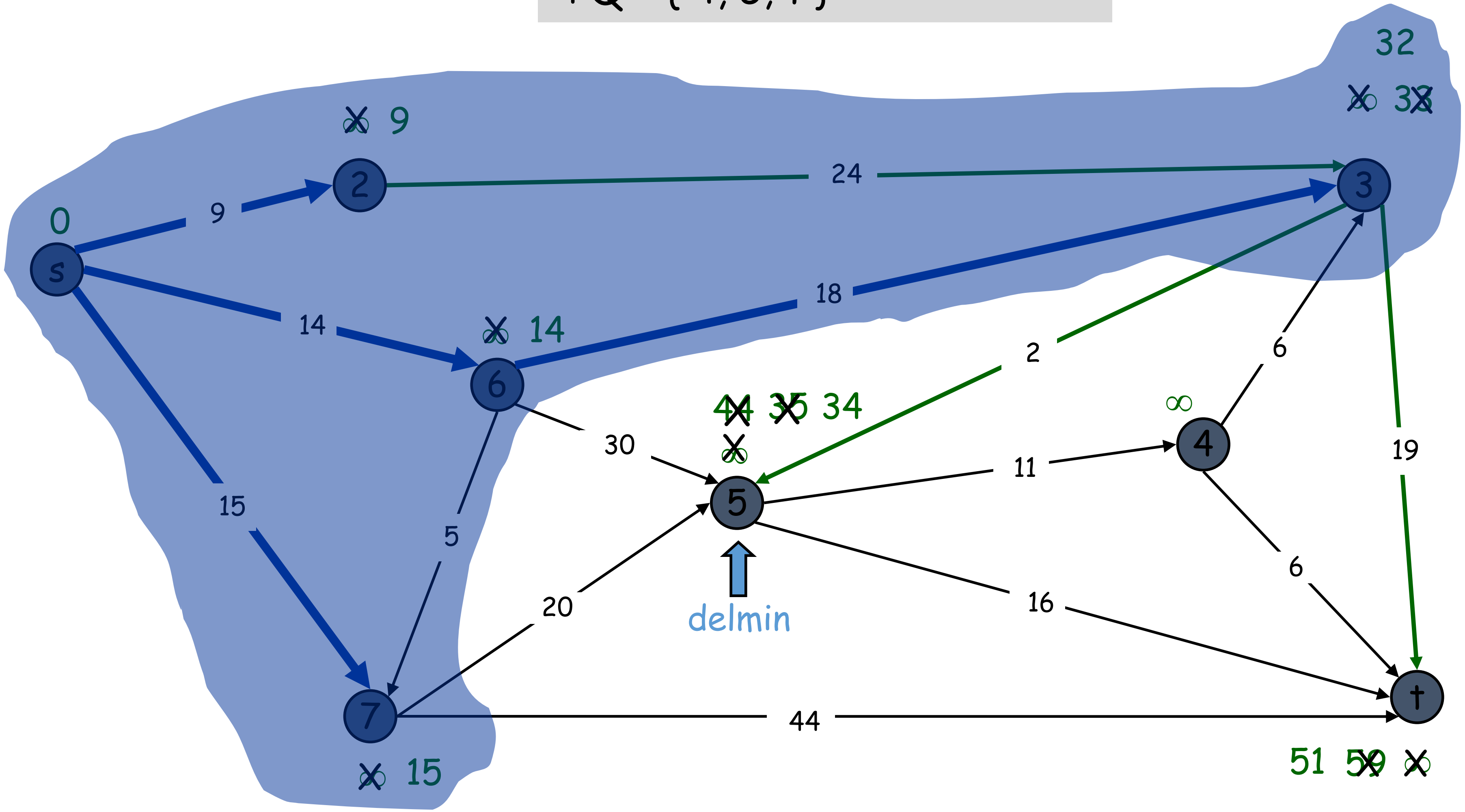
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



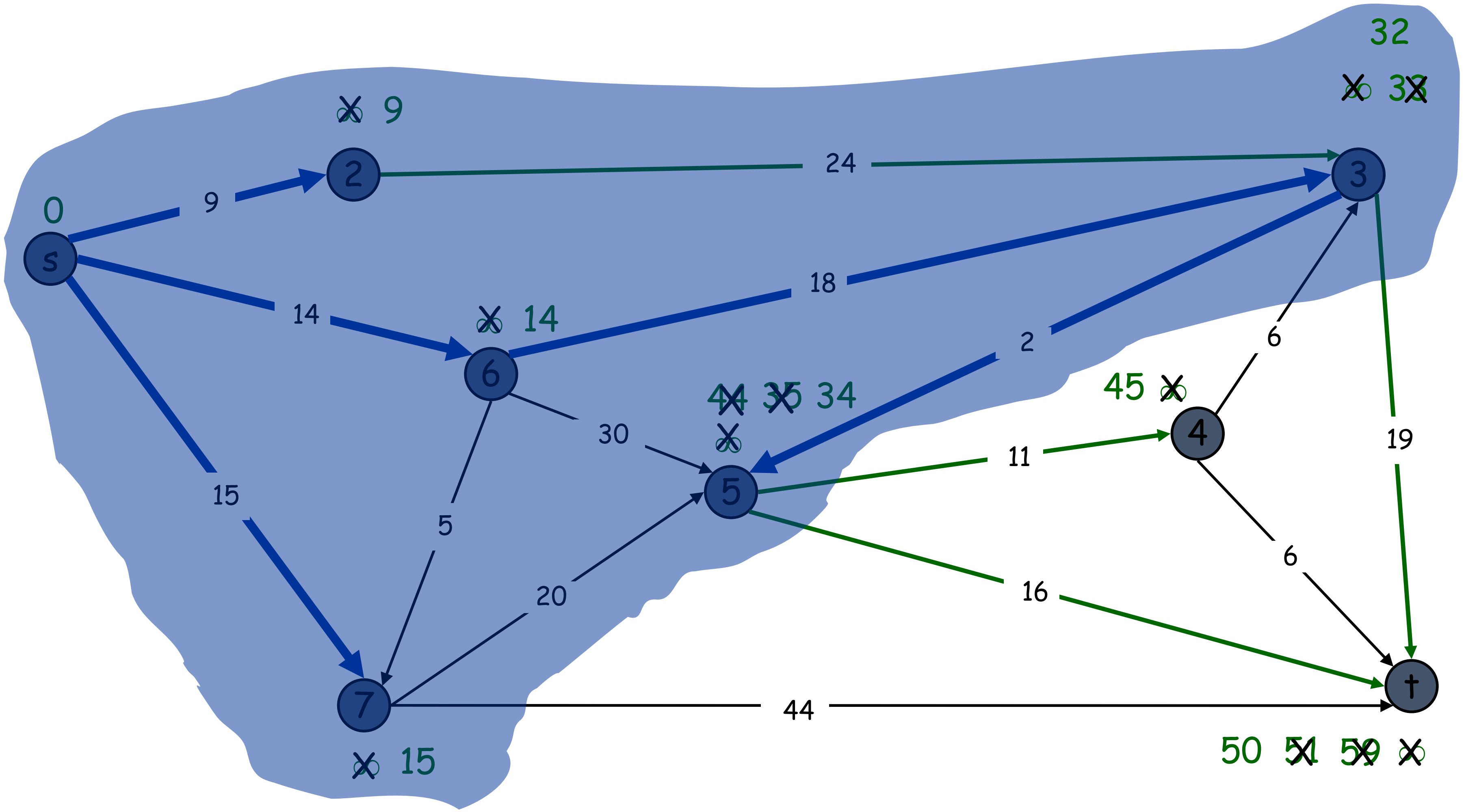
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



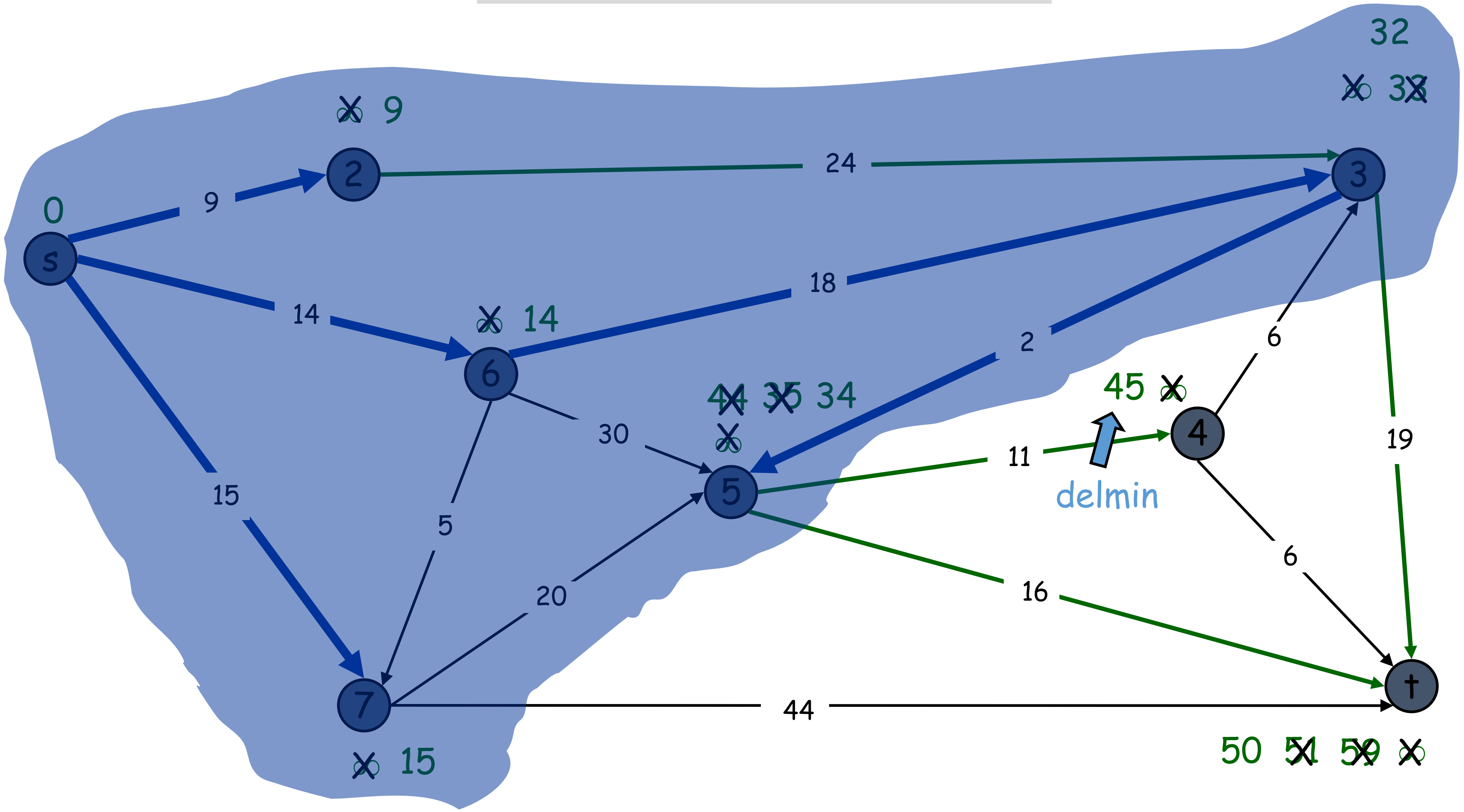
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



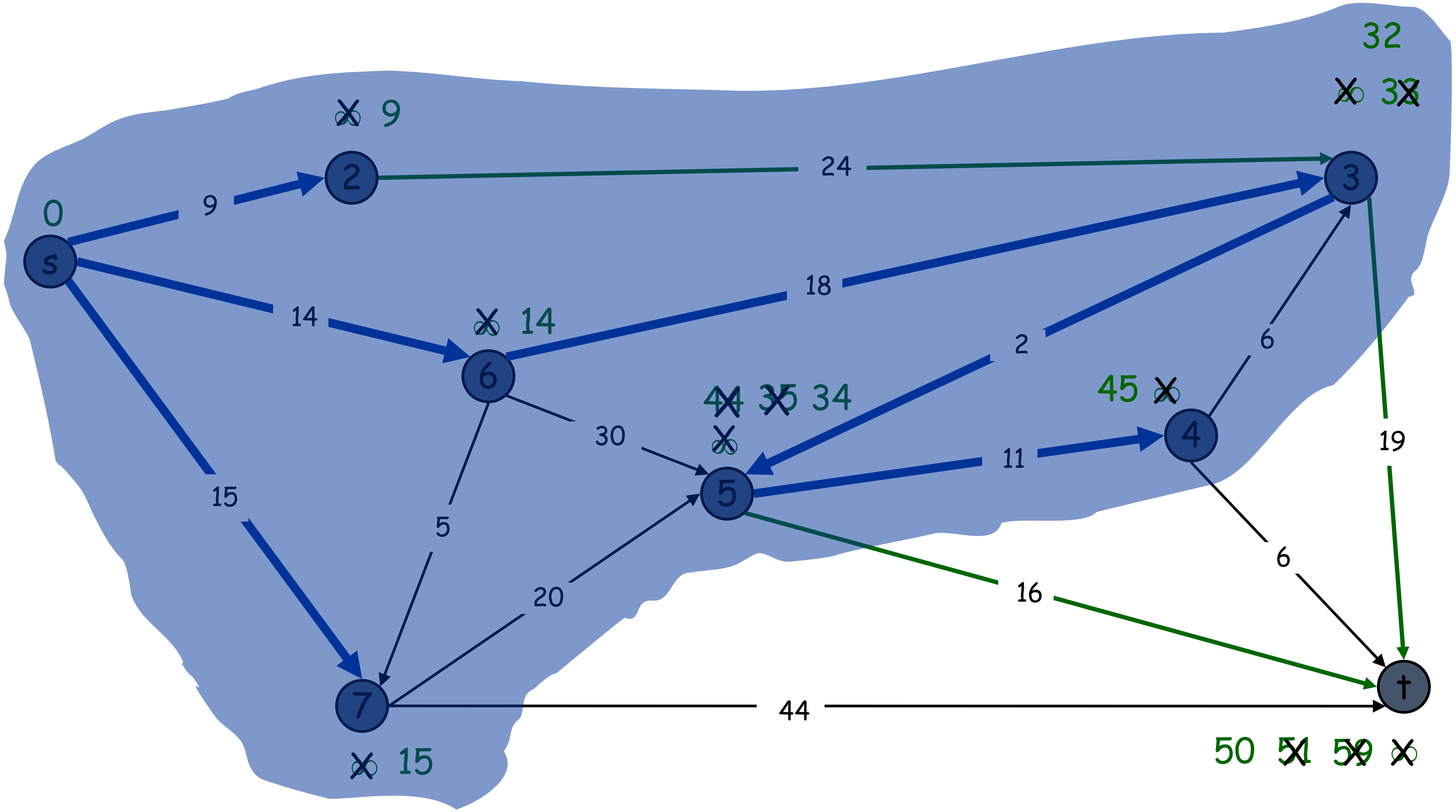
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



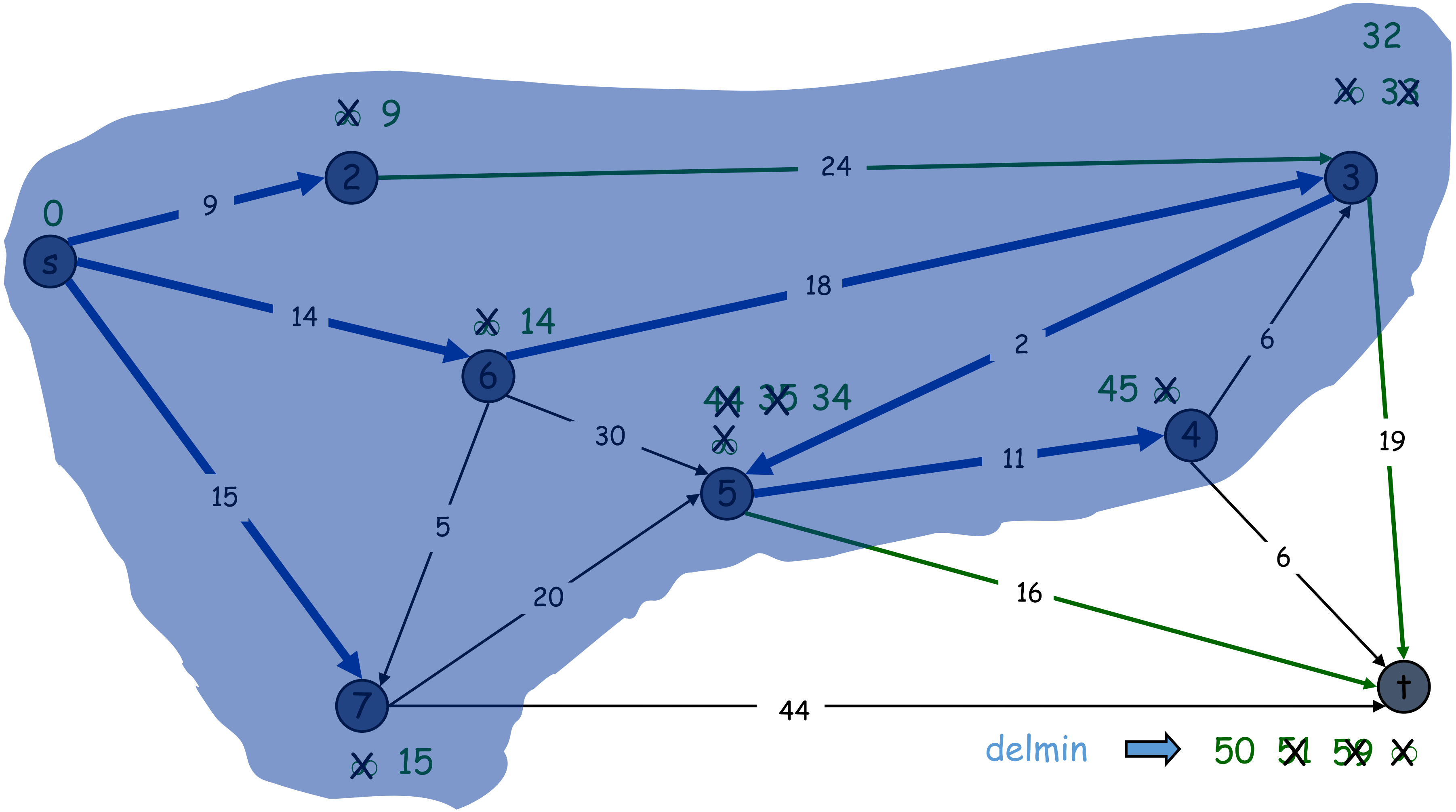
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



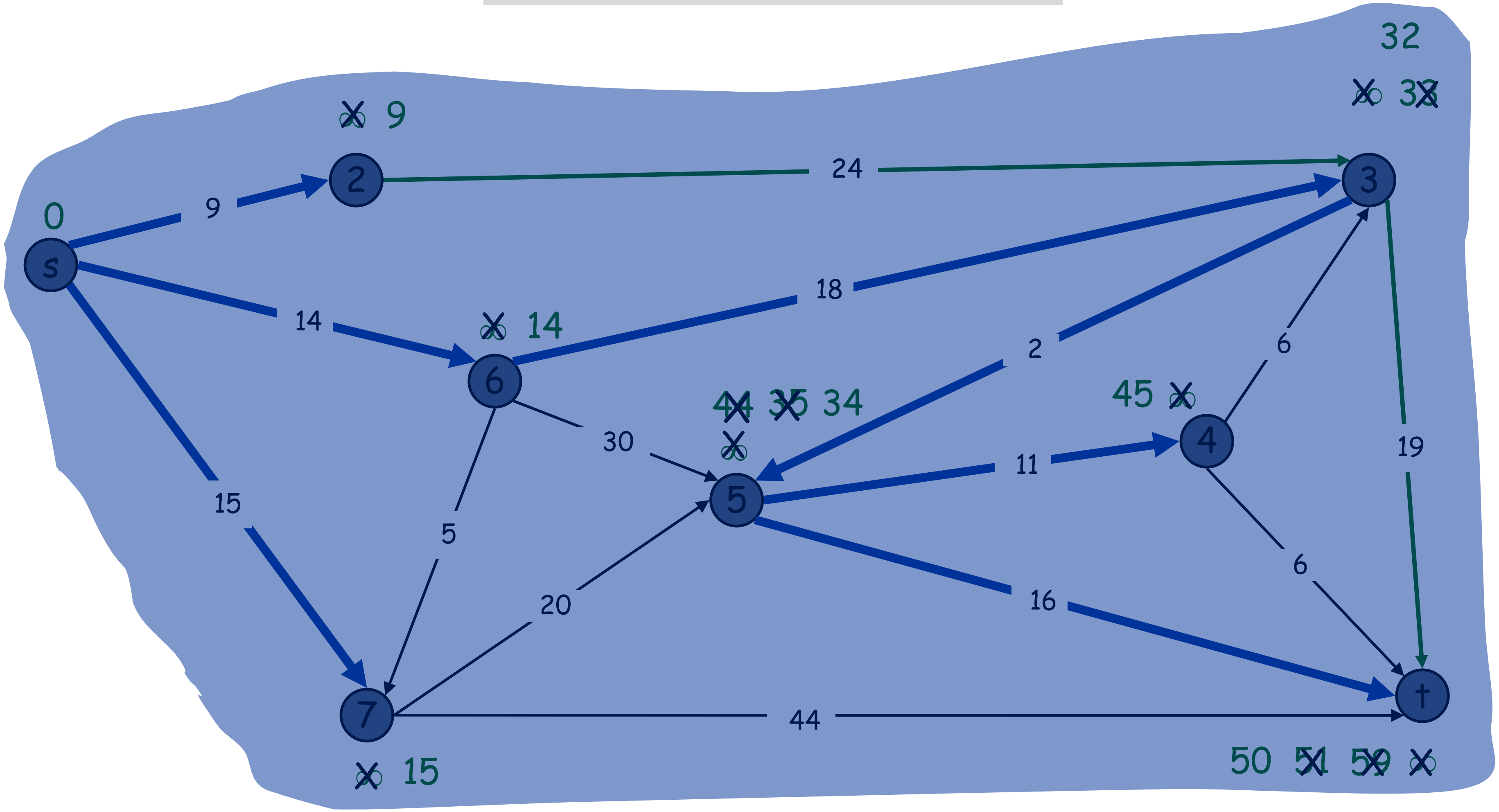
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



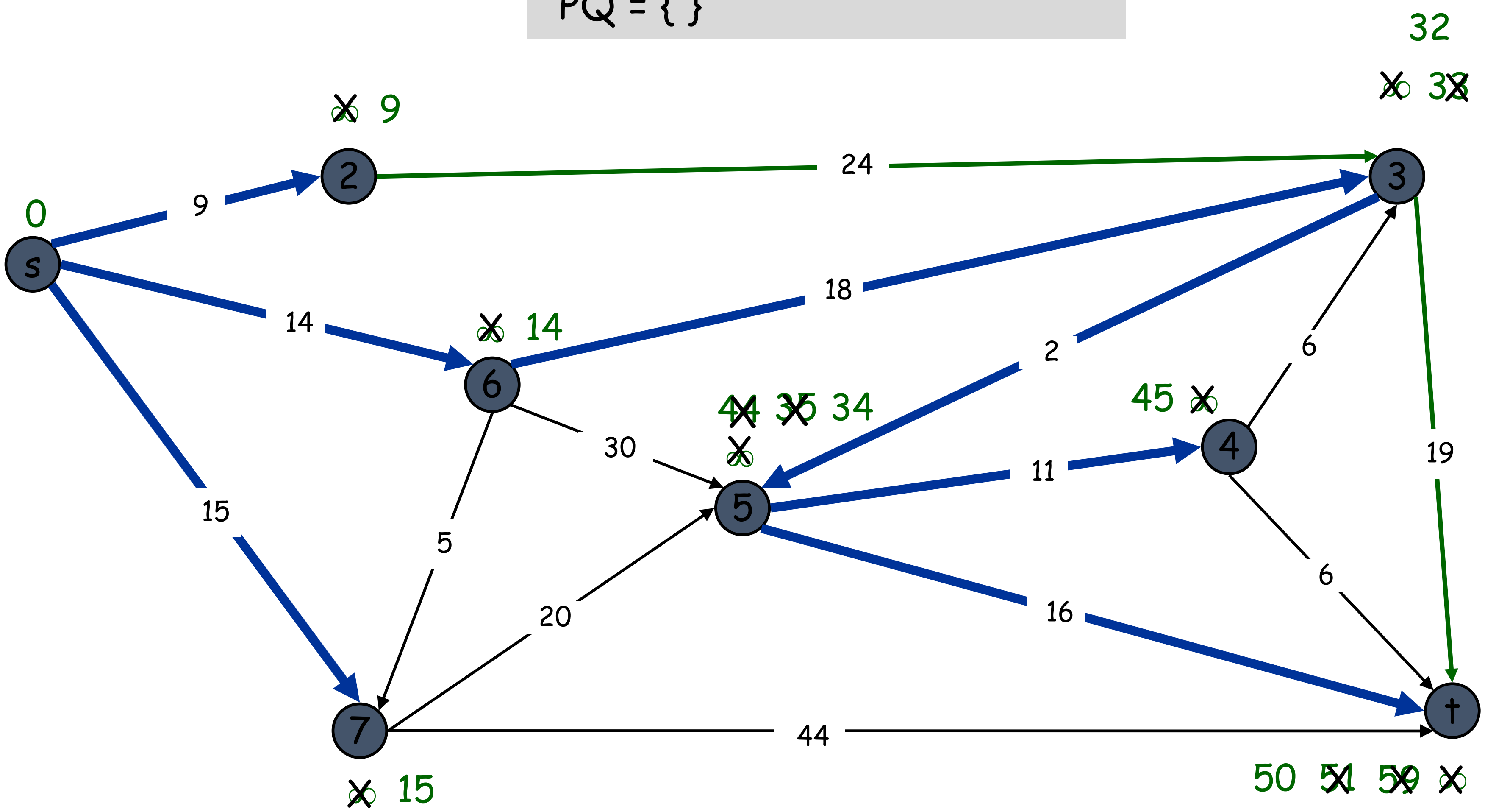
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



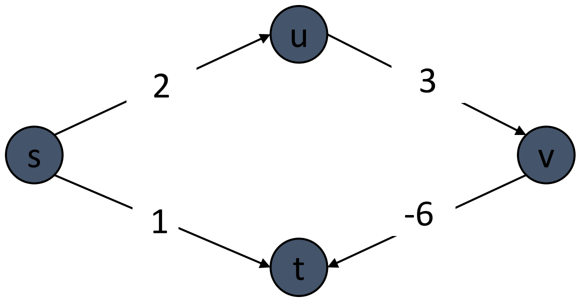
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }

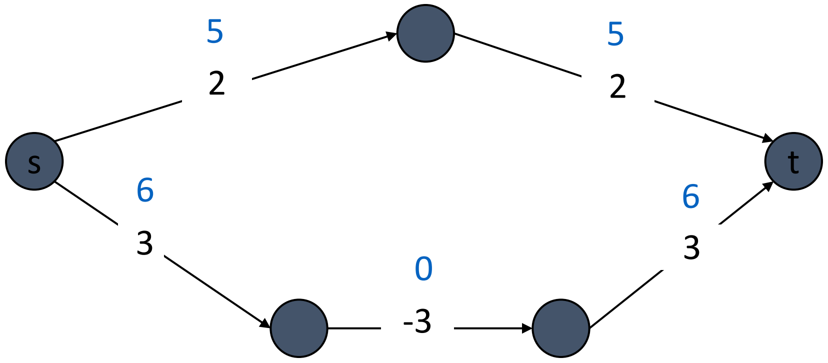


Shortest Paths: Failed Attempts

- Dijkstra. Can fail if negative edge costs.

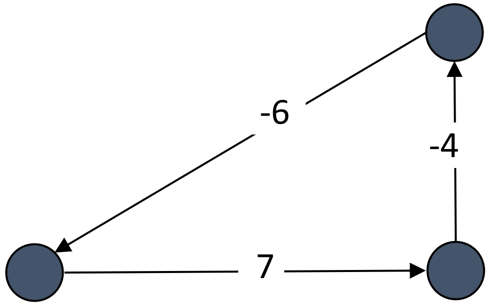


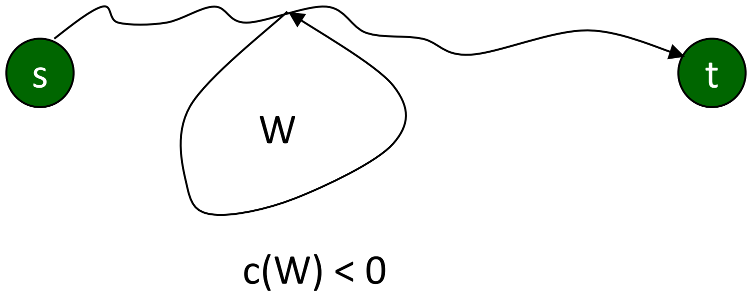
- Re-weighting. Adding a constant to every edge weight can fail.



Shortest Paths: Negative Cost Cycles

- Negative cost cycle.





- Observation. If some path from s to t contains a negative cost cycle, there does not exist a shortest s-t path; otherwise, there exists one that is simple.

Shortest Paths: Dynamic Programming

- Def. OPT(i, v) = length of shortest v-t path P using at most i edges.
  - Case 1: P uses at most i-1 edges.
    - OPT(i, v) = OPT(i-1, v)
  - Case 2: P uses exactly i edges.
    - if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most i-1 edges

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \\ \min\left\{ OPT(i-1, v), \min\limits_{(v,w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

- Remark. By previous observation, if no negative cycles, then OPT(n-1, v) = length of shortest v-t path.

Dynamic Programming

- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

We'll study these with the help of examples.

Dynamic Programming History

- Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

- Etymology.
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"

Dynamic Programming

- Dynamic Programming is a general algorithm design technique
- for solving problems defined by or formulated as recurrences with overlapping subinstances

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS

- “Programming” here means “planning”

- Main idea:
- set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
- - solve smaller instances once
- record solutions in a table
- extract solution to the initial instance from that table

Dynamic Programming: Binary Choice

- Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

  - Case 1: OPT selects job j.
    - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

    optimal substructure

  - Case 2: OPT does not select job j.
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\ \{\ v_j + OPT(p(j)),\ OPT(j-1)\ \} & \text{otherwise} \end{cases}$$

The 0-1 knapsack problem

- A thief breaks into a house, carrying a knapsack...
  - He can carry up to 25 pounds of loot
  - He has to choose which of N items to steal
    - Each item has some weight and some value
    - "0-1" because each item is stolen (1) or not stolen (0)
  - He has to select the items to steal in order to maximize the value of his loot, but cannot exceed 25 pounds
- A greedy algorithm does not find an optimal solution
- A dynamic programming algorithm works well
- This is similar to, but not identical to, the coins problem
  - In the coins problem, we had to make an *exact* amount of change
  - In the 0-1 knapsack problem, we can't *exceed* the weight limit, but the optimal solution may be *less* than the weight limit
  - The dynamic programming solution is similar to that of the coins problem

0/1 Knapsack Problem

- We are given a knapsack of capacity c and a set of n objects numbered $1,2,\ldots,n$. Each object i has weight w_i and profit p_i.
- Let $v = [v_1, v_2, \ldots, v_n]$ be a solution vector in which $v_i = 0$ if object i is not in the knapsack, and $v_i = 1$ if it is in the knapsack.
- The goal is to find a subset of objects to put into the knapsack so that

$$\sum_{i=1}^{n} w_i v_i \leq c$$

(that is, the objects fit into the knapsack) and

$$\sum_{i=1}^{n} p_i v_i$$

is maximized (that is, the profit is maximized).

0/1 Knapsack Problem

- The naive method is to consider all *2^n* possible subsets of the *n* objects and choose the one that fits into the knapsack and maximizes the profit.
- Let *F[i,x]* be the maximum profit for a knapsack of capacity *x* using only objects *{1,2,…,i}*. The DP formulation is:

$$F[i,x] = \begin{cases} 0 & x \geq 0, i = 0 \\ -\infty & x < 0, i = 0 \\ \max\{F[i-1,x], (F[i-1,x-w_i] + p_i)\} & 1 \leq i \leq n \end{cases}$$

0/1 Knapsack Problem

- Construct a table F of size n x c in row-major order.
- Filling an entry in a row requires two entries from the previous row: one from the same column and one from the column offset by the weight of the object corresponding to the row.
- Computing each entry takes constant time; the sequential run time of this algorithm is $\Theta(nc)$.
- The formulation is serial-monadic.

Knapsack Problem

- Knapsack problem.
  - Given n objects and a "knapsack."
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - Goal: fill knapsack so as to maximize total value.

$W = 11$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Ex: { 3, 4 } has value 40.

- Greedy: repeatedly add item with maximum ratio v_i / w_i.
- Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

- Def. OPT(i) = max profit subset of items 1, ..., i.

  - Case 1: OPT does not select item i.
    - OPT selects best of { 1, 2, ..., i-1 }

  - Case 2: OPT selects item i.
    - accepting item i does not immediately imply that we will have to reject other items
    - without knowing what other items were selected before i, we don't even know if we have enough room for i

- Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

- Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

  - Case 1: OPT does not select item i.
    - OPT selects best of { 1, 2, …, i-1 } using weight limit w

  - Case 2: OPT selects item i.
    - new weight limit = w – w_i
    - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } \mathrm{i} = 0 \\ OPT(i-1,w) & \text{if } \mathrm{w_i} > \mathrm{w} \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

- Running time. $\Theta(n\ W)$.
  - Not polynomial in input size!
  - "Pseudo-polynomial."
  - Decision version of Knapsack is NP-complete.
- Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.

Dynamic Programming Summary

- Recipe.
  - Characterize structure of problem.
  - Recursively define value of optimal solution.
  - Compute value of optimal solution.
  - Construct optimal solution from computed information.
- Dynamic programming techniques.
  - Binary choice: weighted interval scheduling.
  - Multi-way choice: segmented least squares.
  - Adding a new variable: knapsack.
  - Dynamic programming over intervals: RNA secondary structure.
- Top-down vs. bottom-up: different people have different intuitions.

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V {
      M[v] ← ∞
      successor[v] ← ϕ
   }

   M[t] = 0
   for i = 1 to n-1 {
      foreach node w ∈ V {
      if (M[w] has been updated in previous iteration) {
         foreach node v such that (v, w) ∈ E {
            if (M[v] > M[w] + c_vw) {
               M[v] ← M[w] + c_vw
               successor[v] ← w
            }
         }
      }
      If no M[w] value changed in iteration i, stop.
   }
}
```

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V {
      M[v] ← ∞
      successor[v] ← ϕ
   }

   M[t] = 0
   for i = 1 to n-1 {
      foreach node w ∈ V {
      if (M[w] has been updated in previous iteration) {
         foreach node v such that (v, w) ∈ E {
            if (M[v] > M[w] + c_vw) {
               M[v] ← M[w] + c_vw
               successor[v] ← w
            }
         }
      }
      If no M[w] value changed in iteration i, stop.
   }
}
```

Detecting Negative Cycles

- Lemma. If OPT(n,v) = OPT(n-1,v) for all v, then no negative cycles.
- Pf. Bellman-Ford algorithm.

- Lemma. If OPT(n,v) < OPT(n-1,v) for some node v, then (any) shortest path from v to t contains a cycle W. Moreover W has negative cost.

- Pf. (by contradiction)
  - Since OPT(n,v) < OPT(n-1,v), we know P has exactly n edges.
  - By pigeonhole principle, P must contain a directed cycle W.
  - Deleting W yields a v-t path with < n edges $\Rightarrow$ W has negative cost.

Searching - Binary search

adds each item in correct place

Find position	$c_1 \log_2 n$
Shuffle down	$c_2\, n$
Overall	$c_1 \log_2 n + c_2\, n$
or	$c_2\, n$

- Each add to the sorted array is *O(n)*

Binary Search: Method

- The method is recursive:
- Compare b with the middle value X[*mid*]
- If b = X[*mid*], return *mid*
- If b < X[*mid*], then b can only be in the left half of X[ ], because X[ ] is sorted. So call the function recursively on the left half.
- If b > X[*mid*], then b can only be in the right half of X[ ], because X[ ] is sorted. So call the function recursively on the right half.

Illustration of Binary search

1	5	7	12	15	20	25	27	35	40	47	60
0	1	2	3	4	5	6	7	8	9	10	11

1	5	7	12	15	20	25	27	35	40	47	60
0	1	2	3	4	5	6	7	8	9	10	11

1	5	7	12	15	20	25	27	35	40	47	60
0	1	2	3	4	5	6	7	8	9	10	11

Binary search code

```
// Returns the index of an occurrence of target in a,
// or a negative number if the target is not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    int min = 0;
    int max = a.length - 1;

    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid;    // target found
        }
    }

    return -(min + 1);    // target not found
}
```

Binary search code

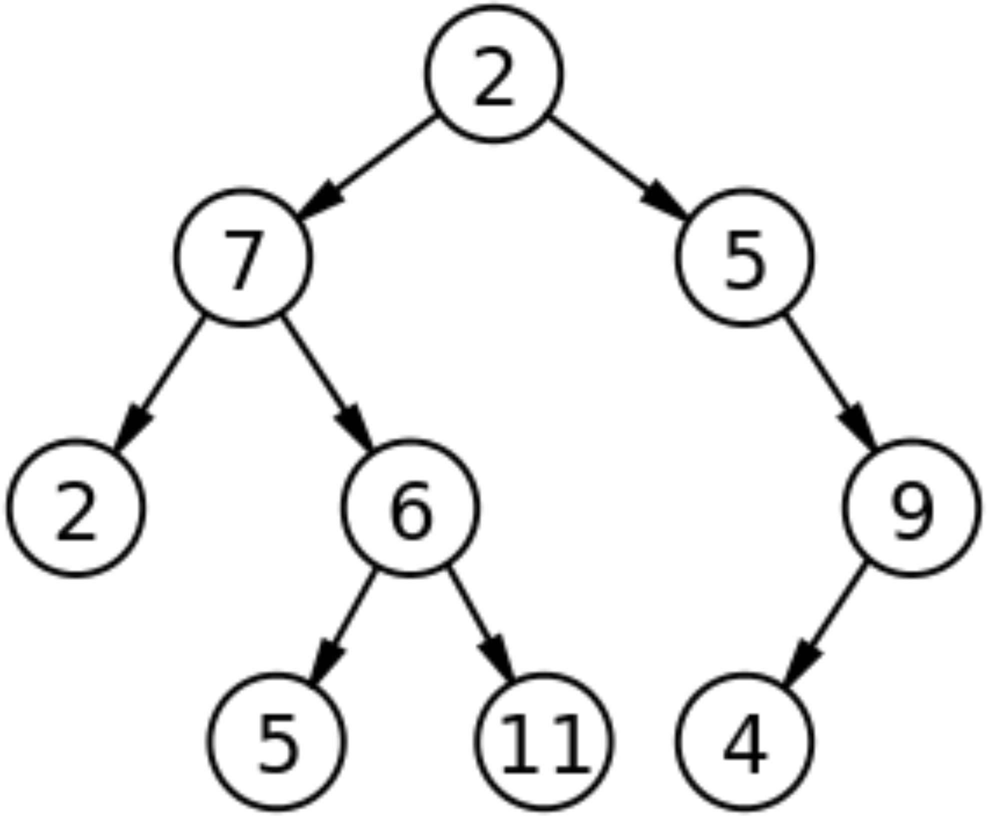
```
// Returns the index of an occurrence of the given value in
// the given array, or a negative number if not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    return binarySearch(a, target, 0, a.length - 1);
}

// Recursive helper to implement search behavior.
private static int binarySearch(int[] a, int target,
                                int min, int max) {
    if (min > max) {
        return -1;          // target not found
    } else {
        int mid = (min + max) / 2;
        if (a[mid] < target) {          // too small; go right
            return binarySearch(a, target, mid + 1, max);
        } else if (a[mid] > target) {   // too large; go left
            return binarySearch(a, target, min, mid - 1);
        } else {
            return mid;     // target found; a[mid] == target
        }
    }
}
```

Time Complexity of Binary Search

- Call $T(n)$ the time of binary search when the array size is n.
- $T(n) = T(n/2) + c$, where c is some constant representing the time of the basis step and the last if-statement to choose between min1 and min2
- Assume for simplicity that $n= 2^k$. (so $k=\log_2 n$)
- $T(2^k)=T(2^{k-1})+c=T(2^{k-2})+c+c=T(2^{k-3})+c+c+c = \ldots = T(2^0)+c+c+\ldots c=T(1)+kc=O(k)=O(\log n)$
- Therefore, $T(n)=O(\log n)$.

Binary Trees

- Binary Tree
  - Consists of
    - Node
    - Left and Right sub-trees
    - Both sub-trees are binary trees



Trees - Implementation

```
struct t_node {
      void *item;
      struct t_node *left;
      struct t_node *right;
      };

typedef struct t_node *Node;

struct t_collection {
      Node root;
      ……
      };
```

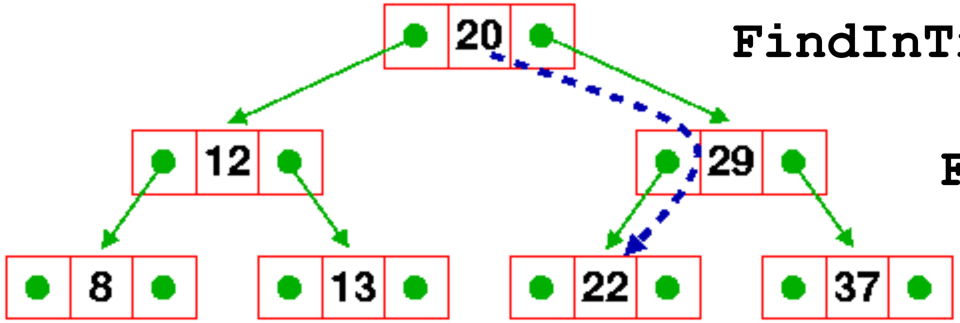
Trees - Implementation

```
extern int KeyCmp( void *a, void *b );
/* Returns -1, 0, 1 for a < b, a == b, a > b */

void *FindInTree( Node t, void *key ) {
    if ( t == (Node)0 ) return NULL;
    switch( KeyCmp( key, ItemKey(t->item) ) ) {
        case -1 : return FindInTree( t->left, key );
        case 0:   return t->item;
        case +1 : return FindInTree( t->right, key );
        }
    }

void *FindInCollection( collection c, void *key ) {
    return FindInTree( c->root, key );
    }
```

Trees - Implementation

- Find
  - key = 22;
    if (FindInCollection(c , &key)) ....

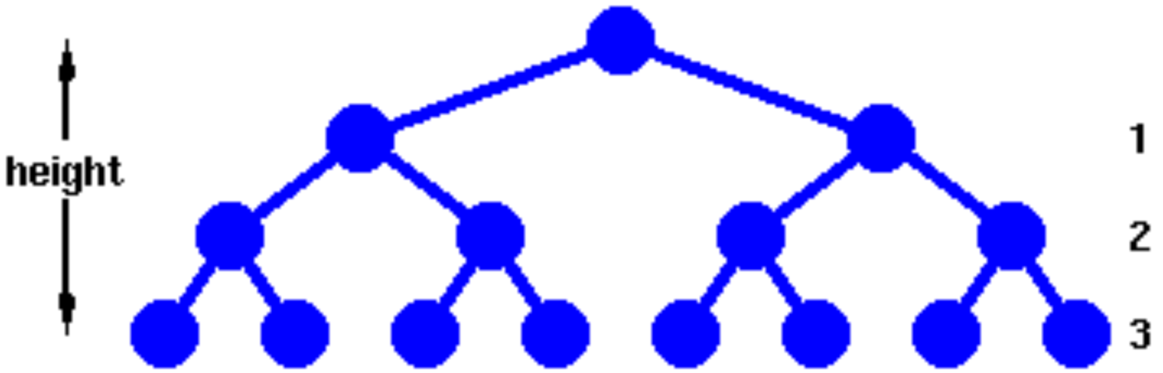
```
n = c->root;
FindInTree( n, &key );
```




```
FindInTree(n->right,&key );
FindInTree(n->left,&key );
return n->item;
```

Trees - Performance

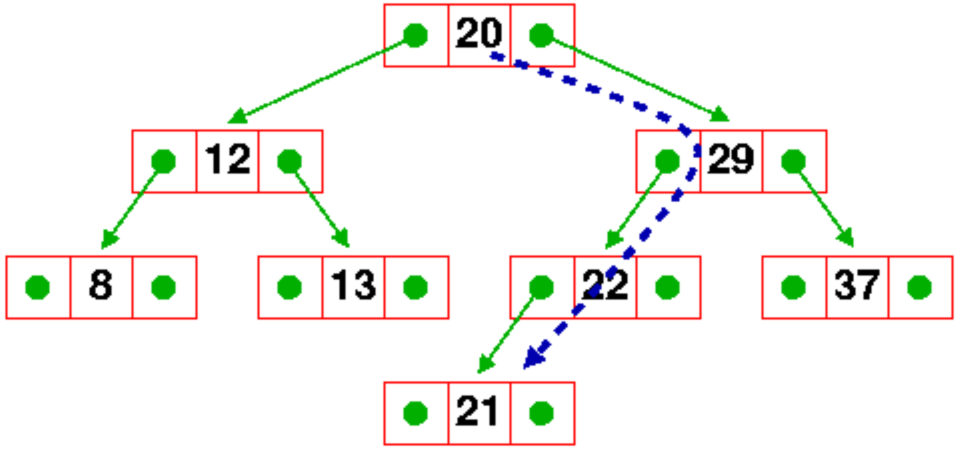
- Find
  - Complete Tree



  - Height, h
    - Nodes traversed in a path from the root to a leaf
  - Number of nodes, h
    - $n = 1 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$
    - $h = \text{floor}(\log_2 n)$

Trees - Addition

- Add 21 to the tree



- We need at most $h+1$ comparisons
- Create a new node (constant time)

$\therefore$ add takes $c_1(h+1)+c_2$ or $c \log n$

- So addition to a tree takes time proportional to $\log n$ also

Trees - Addition - implementation

```
static void AddToTree( Node *t, Node new ) {
   Node base = *t;
   /* If it's a null tree, just add it here */
   if ( base == NULL ) {
      *t = new; return; }
   else
      if( KeyLess(ItemKey(new->item),ItemKey(base->item)) )
         AddToTree( &(base->left), new );
      else
         AddToTree( &(base->right), new );
   }

void AddToCollection( collection c, void *item ) {
        Node new, node_p;
        new = (Node)malloc(sizeof(struct t_node));
        /* Attach the item to the node */
        new->item = item;
        new->left = new->right = (Node)0;
        AddToTree( &(c->node), new );
        }
```

Trees - Addition

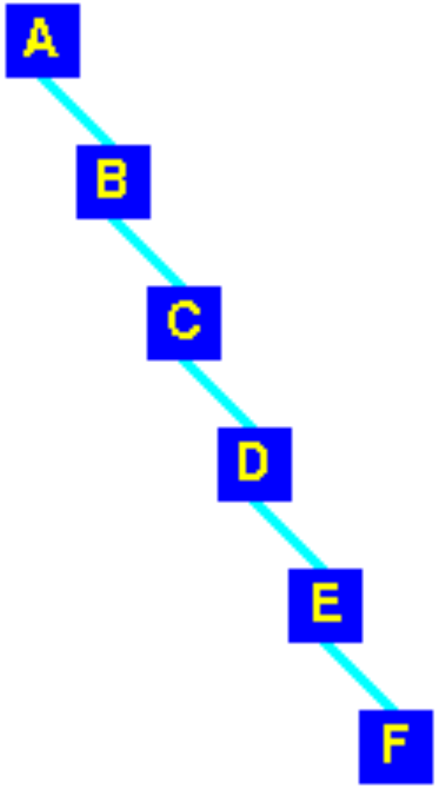
- Find $c \log n$
- Add $c \log n$
- Delete $c \log n$

- Usually efficient in every respect!

- *But there's a catch .... Balance!!!*

Trees - Addition

- Take this list of characters and form a tree
  A B C D E F



- In this case
  - ? Find
  - ? Add
  - ? Delete

Searching - Re-visited

- Binary tree *O(log n) if it stays balanced*
  - Simple binary tree good for static collections
  - Low (preferably zero) frequency of insertions/deletions

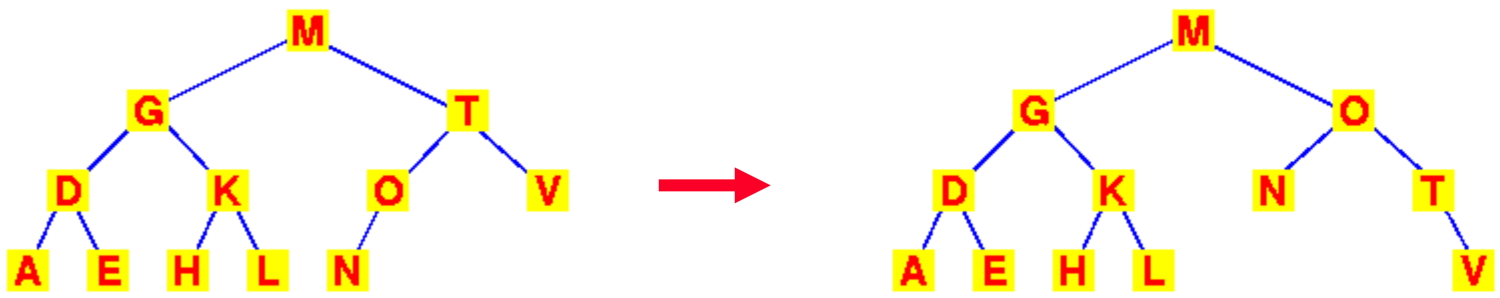
but my collection keeps changing!

  - It's dynamic
  - Need to keep the tree balanced

- First, examine some basic tree operations
  - Useful in several ways!

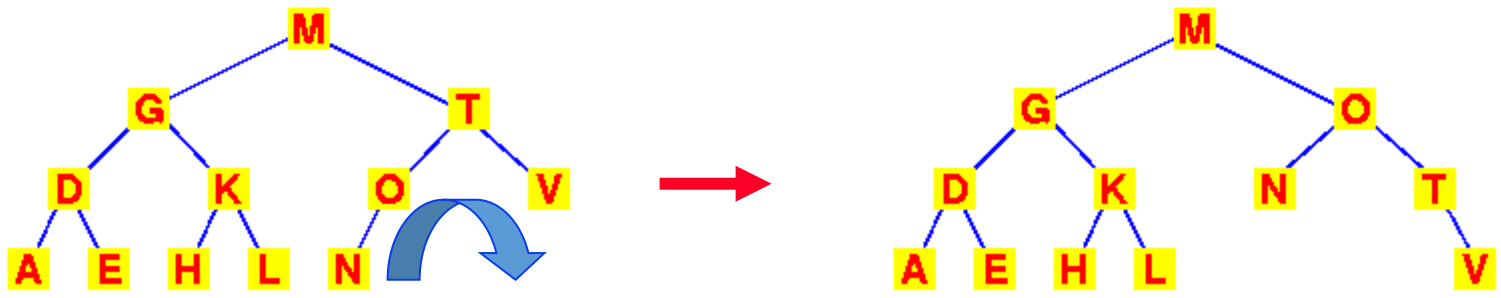
Trees - Searching

- Binary search tree
  - Preserving the order
  - Observe that this transformation preserves the search tree



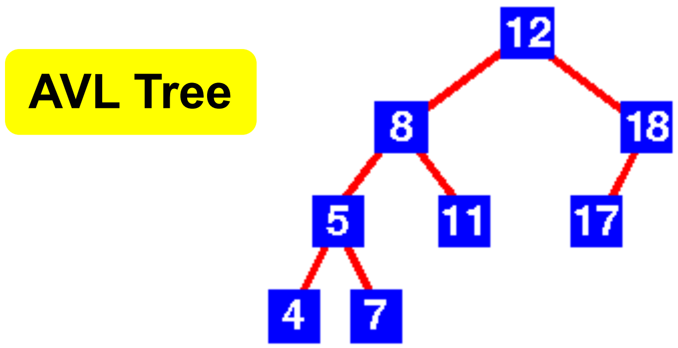
Trees - Searching

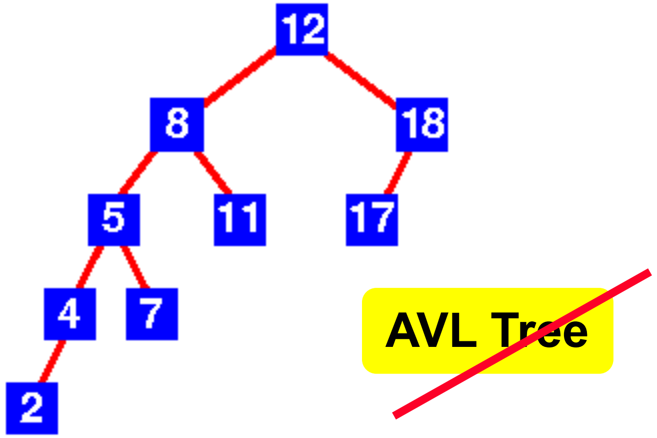
- Binary search tree
  - Preserving the order
  - Observe that this transformation preserves the search tree
- We've performed a rotation of the sub-tree about the T and O nodes



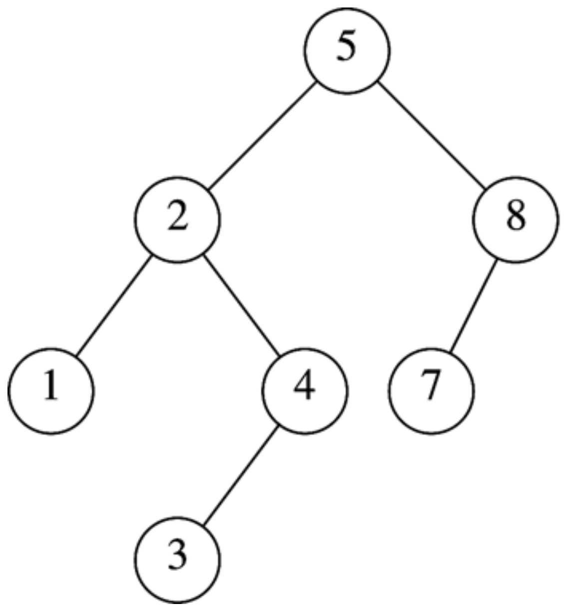
AVL and other balanced trees

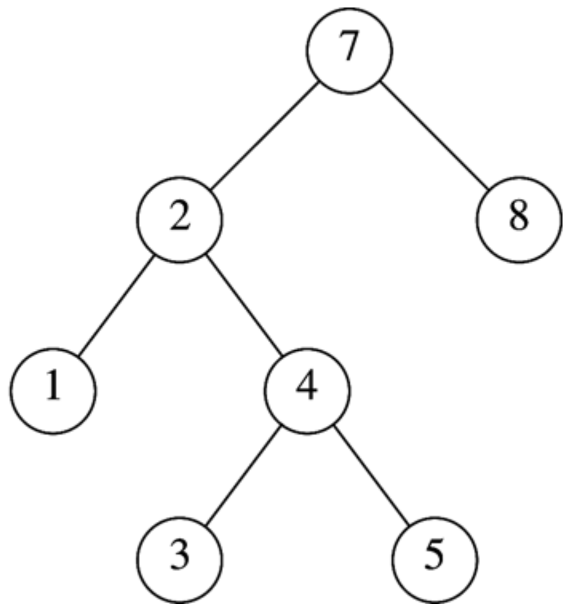
- AVL Trees
  - First balanced tree algorithm
  - Discoverers: Adelson-Velskii and Landis
- Properties
  - Binary tree
  - Height of left and right-subtrees differ by at most 1
  - Subtrees are AVL trees





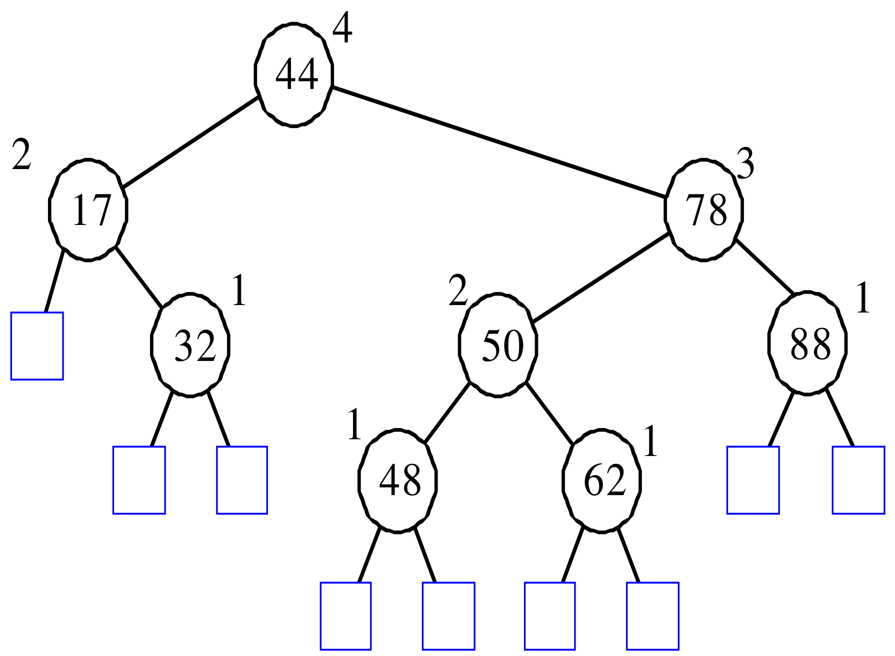
Which is an AVL Tree?





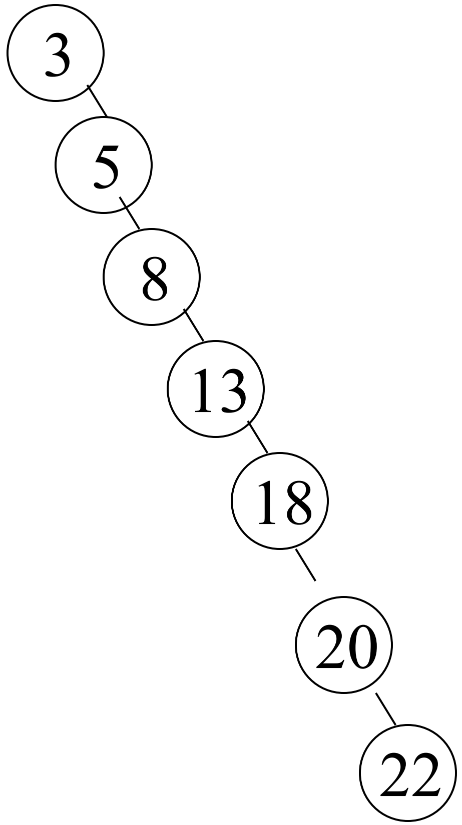
AVL (Adelson-Velskii and Landis) Trees

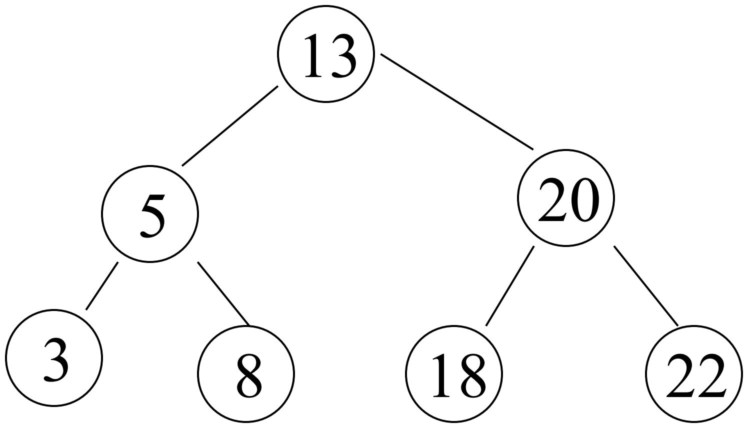
An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1.*



Motivation

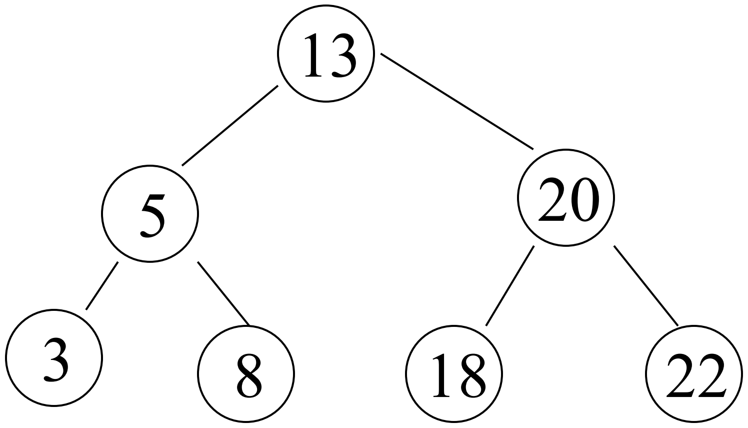
When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22

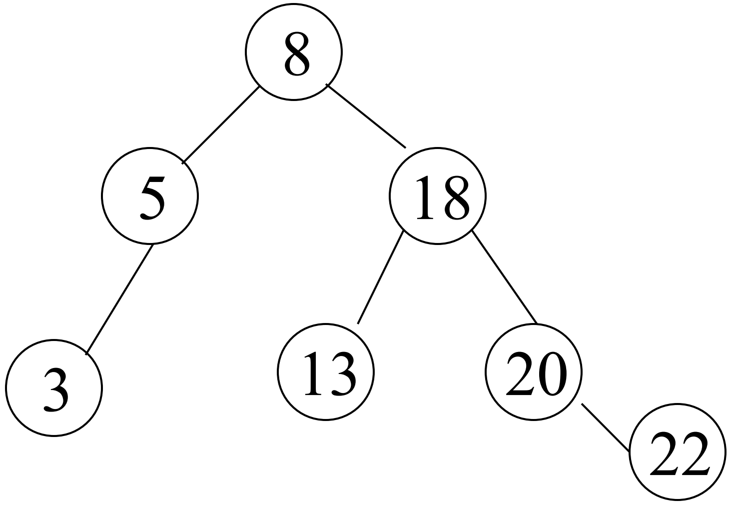




Motivation

- Complete binary tree is hard to build when we allow dynamic insert and remove.
  - We want a tree that has the following properties
    - Tree height = O(log(N))
    - allows dynamic insert and remove with O(log(N)) time complexity.
  - The AVL tree is one of this kind of trees.





AVL (Adelson-Velskii and Landis) Trees

- AVL tree is a binary search tree with balance condition
  - To ensure depth of the tree is O(log(N))
  - And consequently, search/insert/remove complexity bound O(log(N))
- Balance condition
  - For every node in the tree, height of left and right subtree can differ by at most 1

Height of an AVL tree

- Theorem: The ***height*** of an AVL tree storing n keys is O(log n).
- Proof:
  - Let us bound **n(h),** the minimum number of internal nodes of an AVL tree of height h.
  - We easily see that n(0) = 1 and n(1) = 2
  - For h > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2 (at worst).
  - That is, $n(h) >= 1 + n(h-1) + n(h-2)$
  - Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$. So

    $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(n-6)$, … (by induction),

    $n(h) > 2^i n(h-2i)$
  - Solving the base case we get: $n(h) > 2^{h/2-1}$
  - Taking logarithms: $h < 2\log n(h) +2$
  - Since $n>=n(h)$, $h < 2\log(n)+2$ and the height of an AVL tree is O(log n)

AVL Trees - Data Structures

```
typedef enum { LeftHeavy, Balanced, RightHeavy }
                        BalanceFactor;
struct AVL_node {
    BalanceFactor bf;
    void *item;
    struct AVL_node *left, *right;
    }
```

- Insertion
  - Insert a new node (as any binary tree)
  - Work up the tree re-balancing as necessary to restore the AVL property

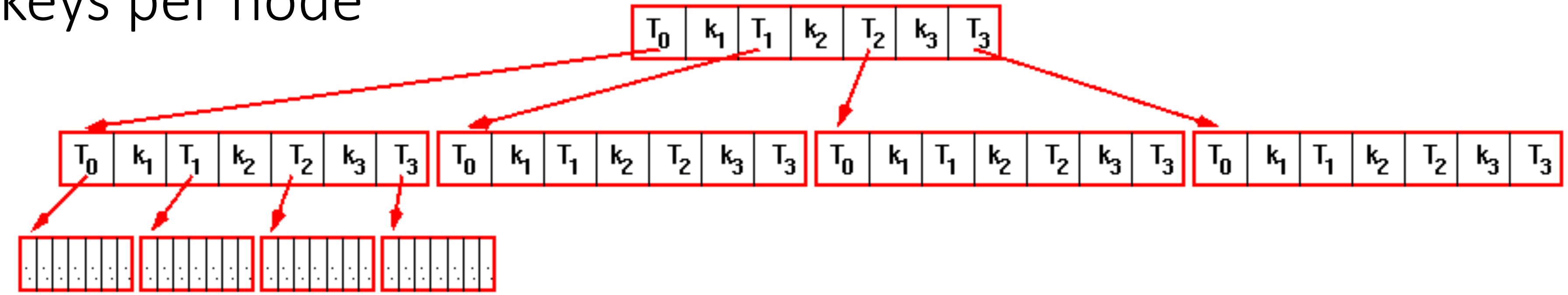
m-way trees (Multiway Trees)

- A multiway tree is a tree that can have more than two children. A multiway tree of order m (or an m-way tree) is one in which a tree can have m children.
- But you have to search through the *m* keys in each node!
- Reduces your gain from having fewer levels.

m-way trees

- Only two children per node?
- Reduce the depth of the tree to $O(\log_m n)$ with *m*-way trees

- *m* children, *m*-1 keys per node



- $m = 10$: 10^6 keys in 6 levels *vs* 20 for a binary tree

B-trees

- All leaves are on the same level
- All nodes except for the root and the leaves have
  - at least *m/2* children
  - at most *m* children
- B+ trees
  - All the keys in the nodes are dummies
  - Only the keys in the leaves point to “real” data
  - Linking the leaves
    - Ability to scan the collection *in order* without passing through the higher nodes

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency

- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions

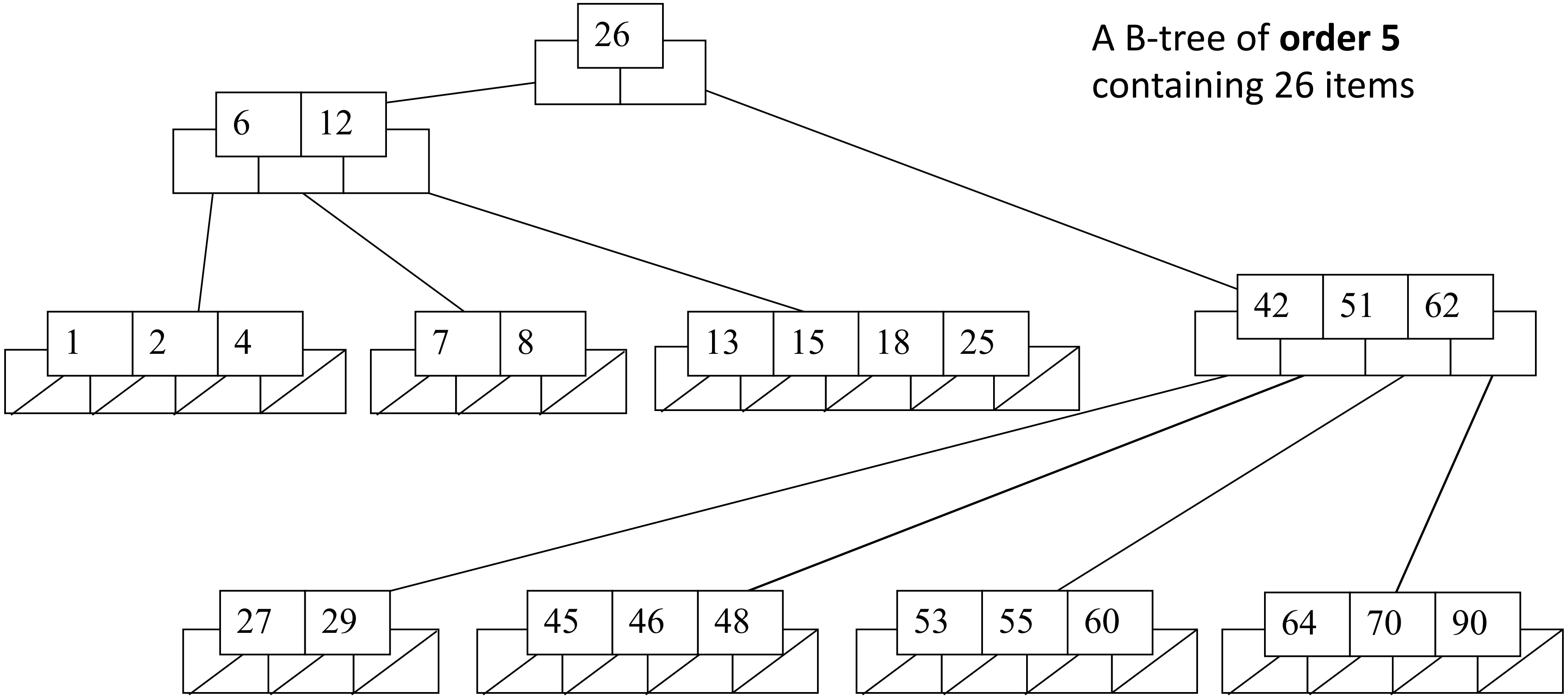
Motivation B-trees

- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 24, so this takes about 0.2 seconds
- We know we can't improve on the log n lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Definition of B-Tree

- Definition assumes external nodes (extended m-way search tree).
- B-tree of order m.
  - m-way search tree.
  - Not empty => root has at least 2 children.
  - Remaining internal nodes (if any) have at least ceil(m/2) children.
  - External (or failure) nodes on same level.

An example B-Tree

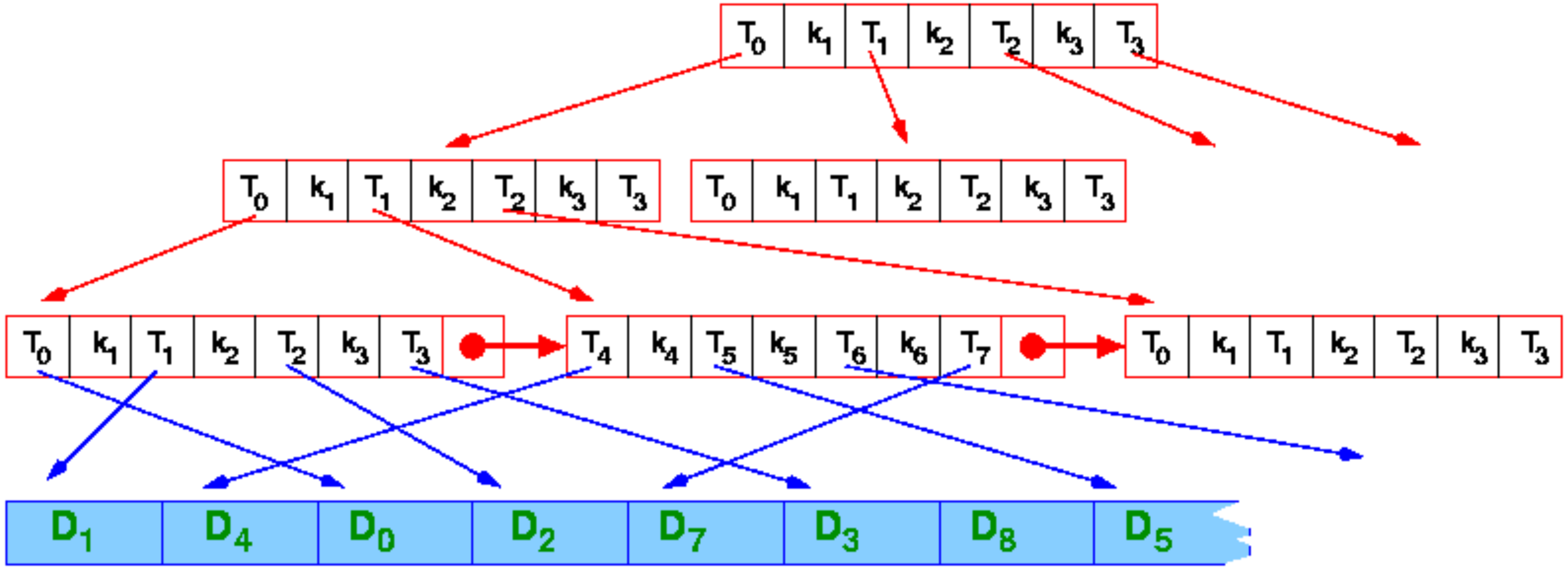


A B-tree of **order 5** containing 26 items

Note that all the leaves are at the same level

B+-trees

- B+ trees
  - All the keys in the nodes are dummies
  - Only the keys in the leaves point to “real” data
  - Data records kept in a separate area

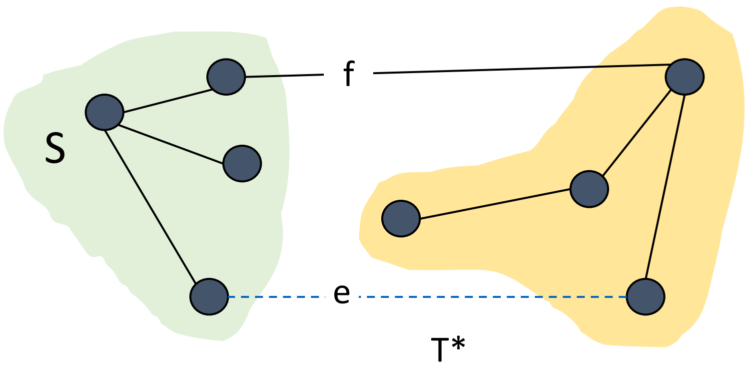


Minimum Spanning Tree (MST)

- A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight

Greedy Algorithms

- Simplifying assumption. All edge costs c_e are distinct.
- Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.
- Pf. (exchange argument)
  - Suppose e does not belong to T*, and let's see what happens.
  - Adding e to T* creates a cycle C in T*.
  - Edge e is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say f, that is in both C and D.
  - $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
  - Since $c_e < c_f$, cost(T') < cost(T*).
  - This is a contradiction. ▪



Prim’s Algorithm

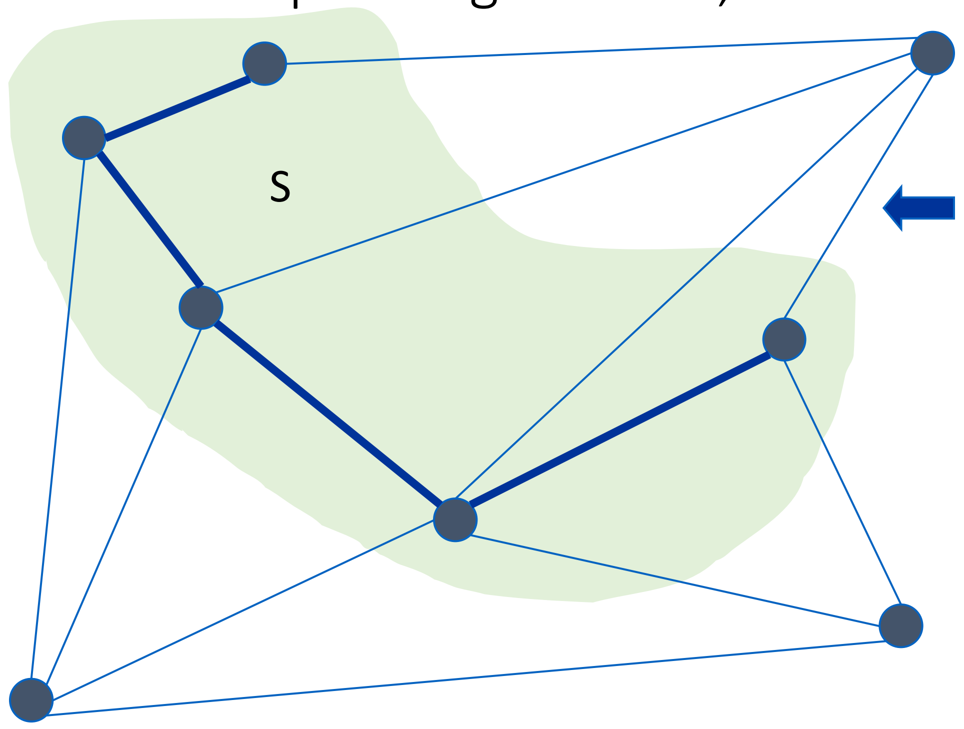
- Initially discovered in 1930 by Vojtěch Jarník, then rediscovered in 1957 by Robert C. Prim
- Similar to Dijkstra’s Algorithm regarding a connected graph
- Starts off by picking any node within the graph and growing from there

Prim’s Algorithm

- Label the starting node, A, with a 0 and all others with infinite
- Starting from A, update all the connected nodes’ labels to A with their weighted edges if it less than the labeled value
- Find the next smallest label and update the corresponding connecting nodes
- Repeat until all the nodes have been visited

Prim's Algorithm: Proof of Correctness

- Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]
  - Initialize S = any node.
  - Apply cut property to S.
  - Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S.



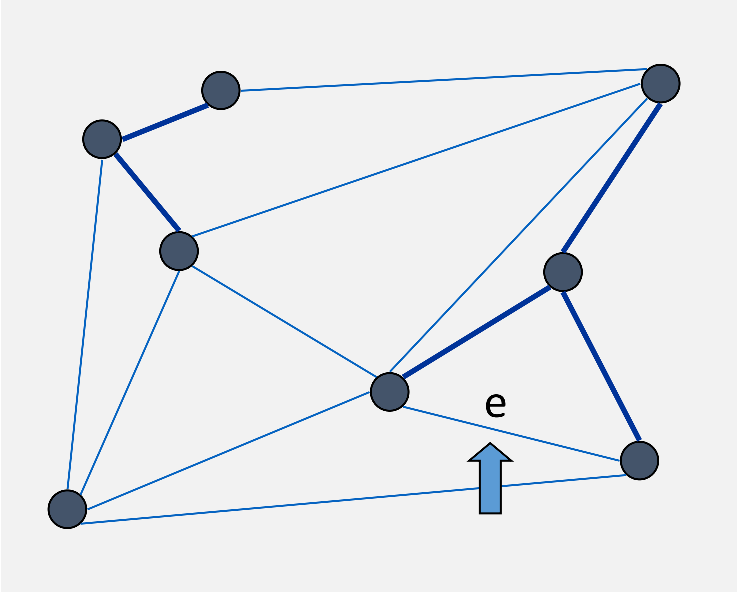
Implementation: Prim's Algorithm

- Implementation. Use a priority queue ala Dijkstra.
  - Maintain set of explored nodes S.
  - For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge v to a node in S.
  - $O(n^2)$ with an array; $O(m \log n)$ with a binary heap.

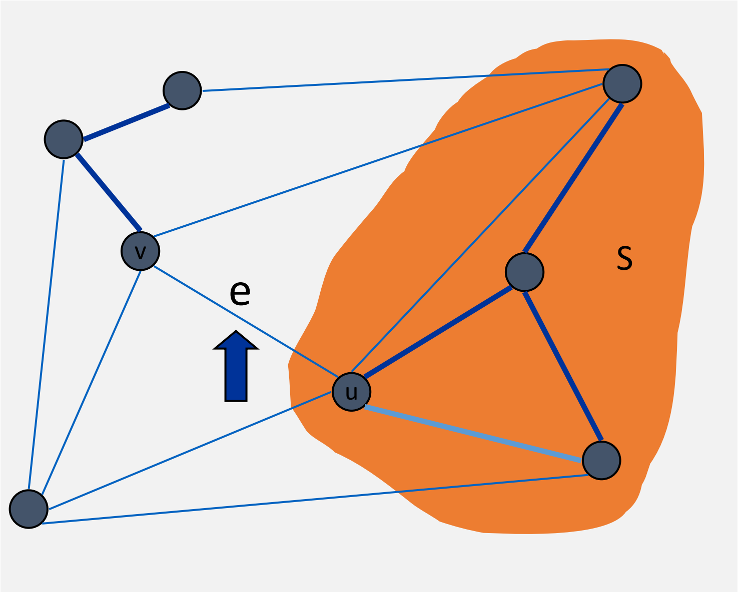
```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← ϕ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
         if ((v ∉ S) and (cₑ < a[v]))
            decrease priority a[v] to cₑ
}
```

Kruskal's Algorithm: Proof of Correctness

- Kruskal's algorithm. [Kruskal, 1956]
  - Consider edges in ascending order of weight.
  - Case 1: If adding e to T creates a cycle, discard e according to cycle property.
  - Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1



Case 2

Kruskal’s Algorithm

- Created in 1957 by Joseph Kruskal
- Finds the MST by taking the smallest weight in the graph and connecting the two nodes and repeating until all nodes are connected to just one tree
- This is done by creating a priority queue using the weights as keys
- Each node starts off as it’s own tree
- While the queue is not empty, if the edge retrieved connects two trees, connect them, if not, discard it
- Once the queue is empty, you are left with the minimum spanning tree

Kruskal's Algorithm

```
KRUSKAL(G):
A = ∅
 foreach v ∈ G.V:
MAKE-SET(v)
foreach (u, v) ordered by weight(u, v), increasing:
  if FIND-SET(u) ≠ FIND-SET(v):
      A = A ∪ {(u, v)}
     UNION(u, v)
return A
```

Implementation: Kruskal's Algorithm

- Implementation. Use the union-find data structure.
  - Build set T of edges in the MST.
  - Maintain set for each connected component.
  - O(m log n) for sorting and O(m α (m, n)) for union-find.

```
Kruskal(G, c) {
   Sort edges weights so that c₁ ≤ c₂ ≤ ... ≤ cₘ.
   T ← φ

   foreach (u ∈ V) make a set containing
singleton u

   for i = 1 to m
      (u,v) = eᵢ
      if (u and v are in different sets) {
         T ← T ∪ {eᵢ}
         merge the sets containing u and v
      }
   return T
}
```

merge two components

Graphs - Kruskal’s Algorithm

- Calculate the minimum spanning tree
  - Put all the vertices into single node trees by themselves
  - Put all the edges in a priority queue
  - Repeat until we’ve constructed a spanning tree
    - *Extract cheapest edge*
    - If it forms a cycle, ignore it
      else add it to the forest of trees
      (it will join two trees into a larger tree)
  - Return the spanning tree

Graphs - Kruskal's Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                                  double **costs ) {
   Forest T;
   Queue q;
   Edge e;
   T = ConsForest( g );
   q = ConsEdgeQueue( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = ExtractCheapestEdge( q );
       } while ( !Cycle( e, T ) );
       AddEdge( T, e );
   }
   return T;
}
```

Graphs - Kruskal's Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
  Forest T;
  Queue q;
  Edge e;
  T = ConsForest( g );
  q = ConsEdgeQueue( g, costs );
  for(i=0;i<(n-1);i++) {
     do {
        e = ExtractCheapestEdge( q );
     } while ( !Cycle( e, T ) );
     AddEdge( T, e );
  }
  return T;
}
```

Graphs - Kruskal's Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                                  double **costs ) {
   Forest T;
   Queue q;
   Edge e;
   T = ConsForest( g );
   q = ConsEdgeQueue( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = ExtractCheapestEdge( q );
       } while ( !Cycle( e, T ) );
       AddEdge( T, e );
   }
   return T;
}
```

Kruskal’s Algorithm

```
Forest MinimumSpanningTree( Graph g, int n,
                                  double **costs ) {
   Forest T;
   Queue q;
   Edge e;
   T = ConsForest( g );
   q = ConsEdgeQueue( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = ExtractCheapestEdge( q );
       } while ( !Cycle( e, T ) );
       AddEdge( T, e );
   }
   return T;
}
```

MST - Time complexity

- Steps
  - Initialise forest $O(\ |V|\ )$
  - Sort edges $O(\ |E| \log |E|\ )$
    - Check edge for cycles $O(\ |V|\ )$ x
    - Number of edges $O(\ |V|\ )$ $O(\ |V|^2\ )$
  - Total $O(\ |V| + |E| \log |E| + |V|^2\ )$
  - Since $|E| = O(\ |V|^2\ )$ $O(\ |V|^2 \log |V|\ )$

  - Thus we would class MST as $O(\ n^2 \log n\ )$ for a graph with n vertices
  - This is an *upper bound*,
    some improvements on this are known ...
    - Prim's Algorithm can be $O(\ |E| + |V| \log |V|\ )$
      using Fibonacci heaps
    - even better variants are known for restricted cases,
      such as sparse graphs ($|E| \gg |V|$)

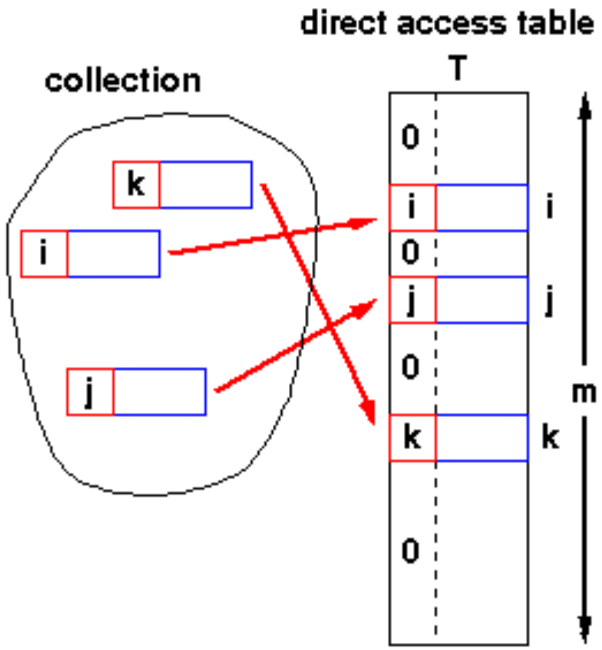
Hash Tables

- All search structures so far
  - Relied on a comparison operation
  - Performance $O(n)$ or $O(\log n)$
- Assume I have a function
  - *f (key)* ® *integer*

    ie one that maps a key to an integer
- What performance might I expect now?

Hash Tables - Structure

- Simplest case:
  - Assume items have integer keys in the range
  - Use the value of the key itself
    to select a slot in a
    direct access table
    in which to store the item
  - To search for an item with key, *k*,
    just look in slot *k*
    - If there's an item there,
      you've found it
    - If the tag is 0, it's missing.
  - Constant time, $O(1)$



Hash Tables - Constraints

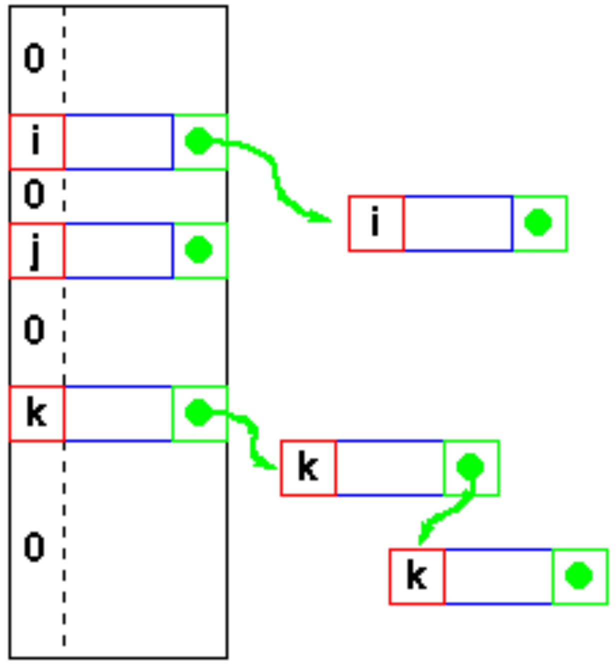
- Constraints
  - Keys must be unique
  - Keys must lie in a small range
  - For storage efficiency,
    keys must be dense in the range
  - If they're sparse (lots of gaps between values),
    a lot of space is used to obtain speed
    - Space for speed trade-off

Hash Tables - Relaxing the constraints

- Keys must be unique
  - Construct a linked list of duplicates “attached” to each slot
  - If a search can be satisfied by *any* item with key, *k*, performance is still *O*(1)

    but
  - If the item has some other distinguishing feature which must be matched, we get $O(n^{max})$

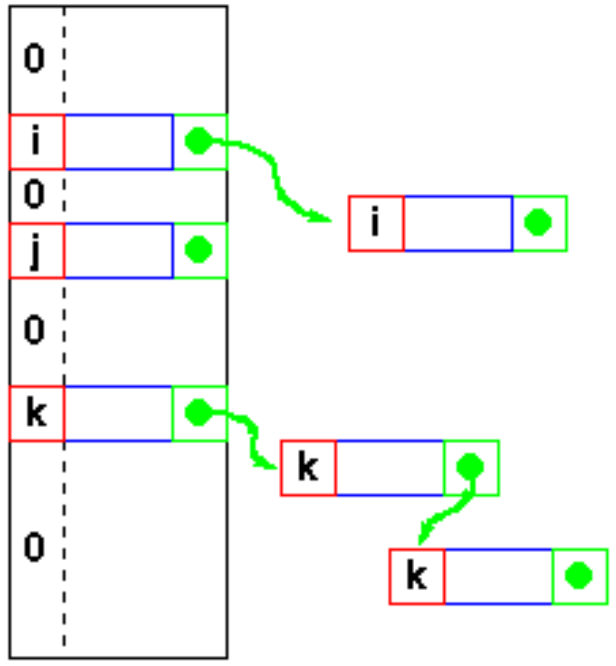
    where n^{max} is the largest number of duplicates - or length of the longest chain



Hash Tables - Relaxing the constraints

- Keys are integers
  - Need a hash function
    h(key) ® *integer*

    ie one that maps a key to an integer
  - Applying this function to the key produces an address
  - If *h* maps each key to a *unique integer* in the range 0 .. *m*-1 then search is *O(1)*



Hash Tables - Hash functions

- Example - using an *n*-character key

```
int hash( char *s, int n ) {
    int sum = 0;
    while( n-- ) sum = sum + *s++;
    return sum % 256;
    }
```

  returns a value in 0 .. 255
- xor function is also commonly used

  sum = sum ^ *s++;
- But any function that generates integers in 0..*m*-1 for some suitable (*not too large*) *m* will do

Hash Tables - Collisions

- Hash function
  - With this hash function

```
int hash( char *s, int n ) {
    int sum = 0;
    while( n-- ) sum = sum + *s++;
    return sum % 256;
    }
```

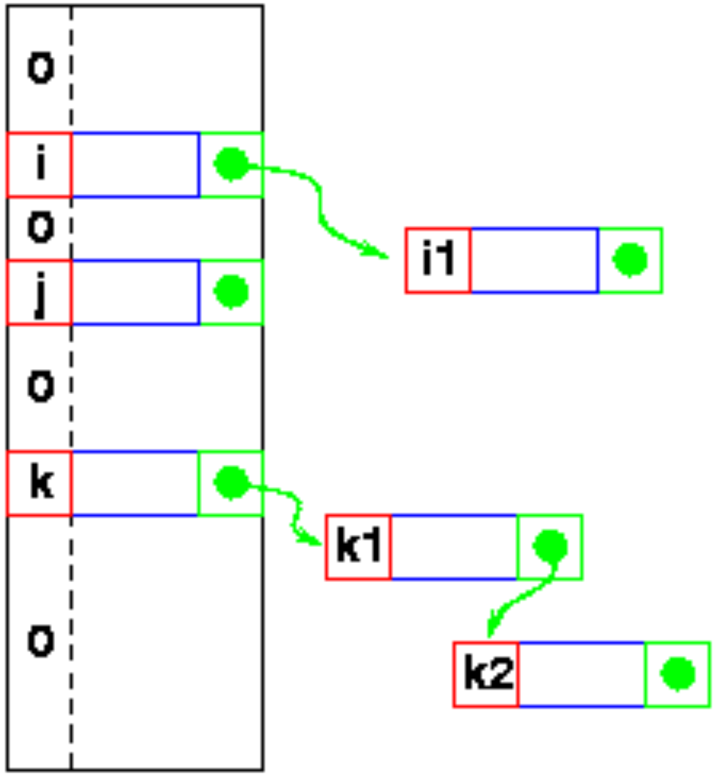
  - hash(“AB”, 2) and
    hash(“BA”, 2)
    return the same value!
  - This is called a collision
  - A variety of techniques are used for resolving collisions

Hash Tables - Collision handling

- Collisions
  - Occur when the hash function maps two different keys to the same address
  - The table must be able to recognise and resolve this
  - Recognise
    - Store the actual key with the item in the hash table
    - Compute the address
      - k = h(key)
    - Check for a hit
      - *if (* table[k].key == key *) then hit*
        else try next entry
  - Resolution
    - Variety of techniques

Hash Tables - Linked lists

- Collisions - Resolution
  - ❶Linked list attached
    to each primary table slot
    - h(i) == h(i1)
    - h(k) == h(k1) == h(k2)
  - Searching for i1
    - Calculate h(i1)
    - Item in table, i,
      doesn't match
    - Follow linked list to i1
  - If NULL found,
    key isn't in table



Hash Tables - Overflow area

Overflow area

- Linked list constructed in special area of table called overflow area

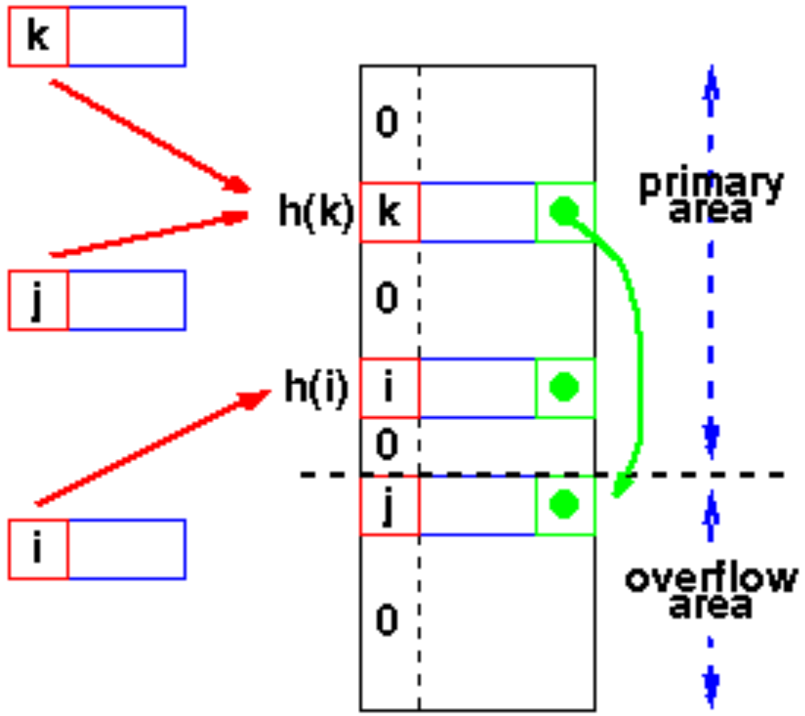
h(k) == h(j)

k stored first

Adding j

- Calculate h(j)
- Find k
- Get first slot in overflow area
- Put j in it
- k's pointer points to this slot

Searching - same as linked list



Hash Tables - Re-hashing

Use a second hash function

- Many variations
- General term: re-hashing

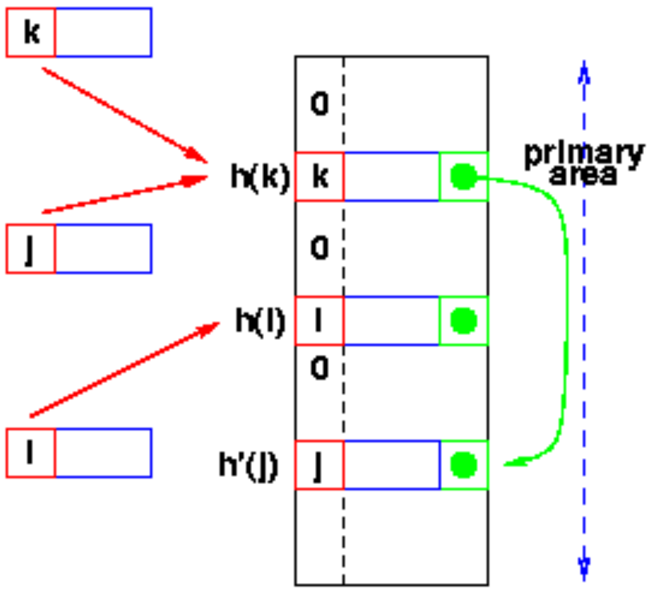
$h(k) == h(j)$

k stored first

Adding j

- Calculate $h(j)$
- Find k
- Repeat until we find an empty slot
  - Calculate $h'(j)$
- Put j in it

Searching - Use $h(x)$, then $h'(x)$



Hash Tables - Re-hash functions

The re-hash function

Many variations

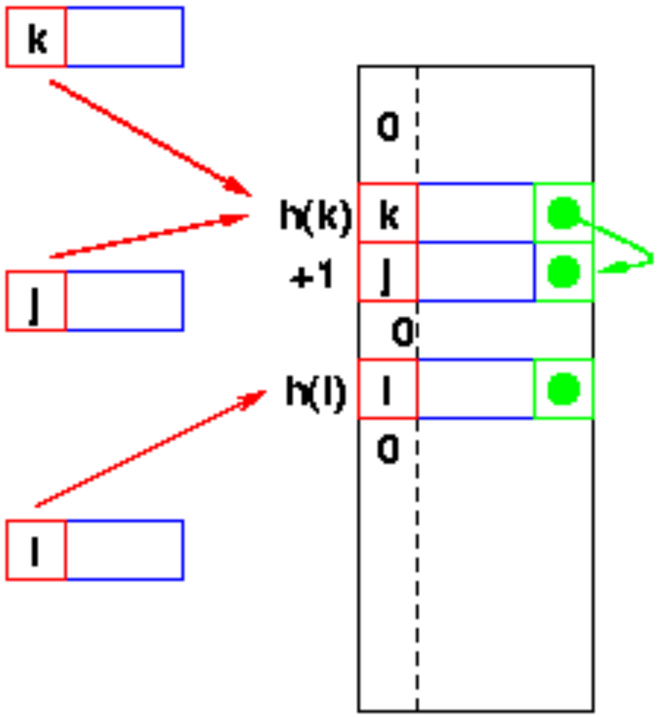
Linear probing

h'(x) is +1

Go to the next slot
until you find one empty

Can lead to bad clustering

Re-hash keys fill in gaps
between other keys and exacerbate
the collision problem



Hash Tables - Summary so far ...

- Potential *O(1)* search time
  - If a suitable function *h(key)* ® *integer* can be found
- Space for speed trade-off
  - "Full" hash tables don't work (more later!)
- Collisions
  - Inevitable
    - Hash function reduces amount of information in key
  - Various resolution strategies
    - Linked lists
    - Overflow areas
    - Re-hash functions
      - Linear probing h' is $+1$
      - Quadratic probing h' is $+ci^2$
      - Any other hash function!
        - or even sequence of functions!

Hash Tables - Choosing the Hash Function

- “Almost any function will do”
  - But some functions are definitely better than others!
- Key criterion
  - Minimum number of collisions
    - Keeps chains short
    - Maintains $O(1)$ average

Collision Frequency

- Birthdays *or* the von Mises paradox
  - There are 365 days in a normal year
    - ➧ Birthdays on the same day unlikely?
  - How many people do I need before "it's an even bet" (*ie* the probability is > 50%) that two have the same birthday?
  - View
    - the days of the year as the slots in a hash table
    - the "birthday function" as mapping people to slots
  - Answering von Mises' question answers the question about the probability of collisions in a hash table

Birthday Problem

- What is the smallest number of people you need in a group so that the probability of 2 or more people having the same birthday is greater than 1/2?
- Answer: 23

No. of people	23	30	40	60
Probability	.507	.706	.891	.994

Birthday Problem

- A={at least 2 people in the group have a common birthday}
- A' = {no one has common birthday}

$$3 \ \ people \quad : P(A') = \frac{364}{365} \times \frac{363}{365}$$

$$23 \ \ people \quad :$$

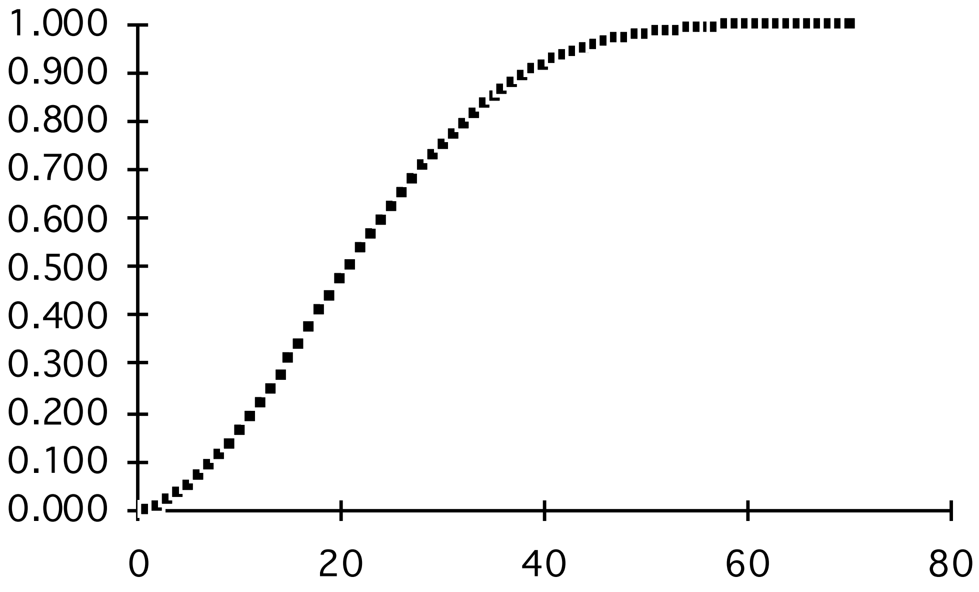
$$P(A') = \frac{364}{365} \times \frac{363}{365} \times \dots \frac{343}{365} = .498$$

$$so \ \ P(A) = 1 - P(A') = 1 - .498 \ = .502$$

Coincident Birthdays

- Probability of having two identical birthdays
- $P(n) = 1 - Q(n)$
- $P(23) = 0.507$

- With 23 entries, table is only 23/365 = 6.3% full!



Decision Problems

- Decision problem.
  - X is a set of strings.
  - Instance: string s.
  - Algorithm A solves problem X: A(s) = yes iff s $\in$ X.

length of s

- Polynomial time. Algorithm A runs in poly-time if for every string s, A(s) terminates in at most p(|s|) "steps", where p(·) is some polynomial.
- PRIMES: X = { 2, 3, 5, 7, 11, 13, 17, 23, 29, 31, 37, ... }

Definition of P

- P. Decision problems for which there is a poly-time algorithm.

Problem	Description	Algorithm	Yes	No
MULTIPLE	Is x a multiple of y?	Grade school division	51, 17	51, 16
RELPRIME	Are x and y relatively prime?	Euclid (300 BCE)	34, 39	34, 51
PRIMES	Is x prime?	AKS (2002)	53	51
EDIT-DISTANCE	Is the edit distance between x and y less than 5?	Dynamic programming	niether neither	acgggt ttttta
LSOLVE	Is there a vector x that satisfies Ax = b?	Gauss-Edmonds elimination	$\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix}, \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

NP

- Certification algorithm intuition.
  - Certifier views things from "managerial" viewpoint.
  - Certifier doesn't determine whether $s \in X$ on its own; rather, it checks a proposed proof t that $s \in X$.

- Def. Algorithm $C(s, t)$ is a certifier for problem X if for every string s, $s \in X$ iff there exists a string t such that $C(s, t)$ = yes.

- NP. Decision problems for which there exists a poly-time certifier.

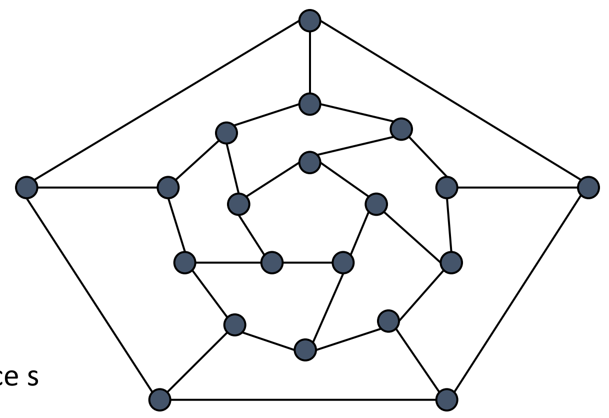
- Remark. NP stands for nondeterministic polynomial-time.

Certifiers and Certificates: Hamiltonian Cycle

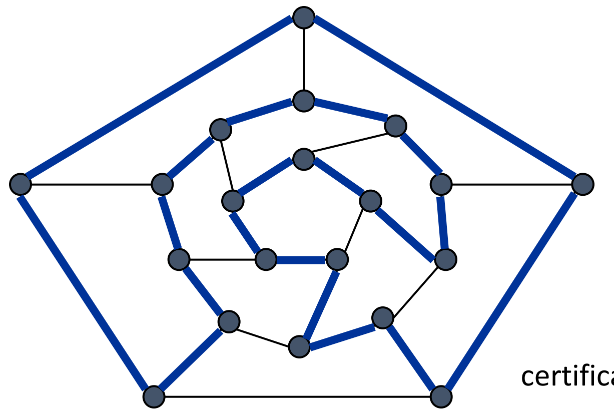
HAM-CYCLE. Given an undirected graph G = (V, E), does there exist a simple cycle C that visits every node?

Certificate. A permutation of the n nodes.

- Certifier. Check that the permutation contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.
- Conclusion. HAM-CYCLE is in NP.



instance s



certificate t

P, NP, EXP

- P. Decision problems for which there is a poly-time algorithm.
- EXP. Decision problems for which there is an exponential-time algorithm.
- NP. Decision problems for which there is a poly-time certifier.
- Claim. P $\subseteq$ NP.
- Pf. Consider any problem X in P.
  - By definition, there exists a poly-time algorithm A(s) that solves X.
  - Certificate: $t = \varepsilon$, certifier C(s, t) = A(s). ▪
- Claim. NP $\subseteq$ EXP.
- Pf. Consider any problem X in NP.
  - By definition, there exists a poly-time certifier C(s, t) for X.
  - To solve input s, run C(s, t) on all strings t with $|t| \leq p(|s|)$.
  - Return yes, if C(s, t) returns yes for any of these. ▪

Polynomial Transformation (Mapping)

- Def. Problem X polynomial reduces (Cook) to problem Y if arbitrary instances of problem X can be solved using:
  - Polynomial number of standard computational steps, plus
  - Polynomial number of calls to oracle that solves problem Y.

- Def. Problem X polynomial transforms (Karp) to problem Y if given any input x to X, we can construct an input y such that x is a yes instance of X iff y is a yes instance of Y.

- Note. Polynomial transformation is polynomial reduction with just one call to oracle for Y, exactly at the end of the algorithm for X. Almost all previous reductions were of this form.

- Open question. Are these two concepts the same?

NP-Complete

- NP-complete. A problem Y in NP with the property that for every problem X in NP, $X \leq_p Y$.

- Theorem. Suppose Y is an NP-complete problem. Then Y is solvable in poly-time iff P = NP.
- Pf. $\Leftarrow$ If P = NP then Y can be solved in poly-time since Y is in NP.
- Pf. $\Rightarrow$ Suppose Y can be solved in poly-time.
  - Let X be any problem in NP. Since $X \leq_p Y$, we can solve X in poly-time. This implies NP $\subseteq$ P.
  - We already know P $\subseteq$ NP. Thus P = NP. ▪

- Fundamental question. Do there exist "natural" NP-complete problems?

Polynomial-Time Reduction

- Purpose. Classify problems according to relative difficulty.

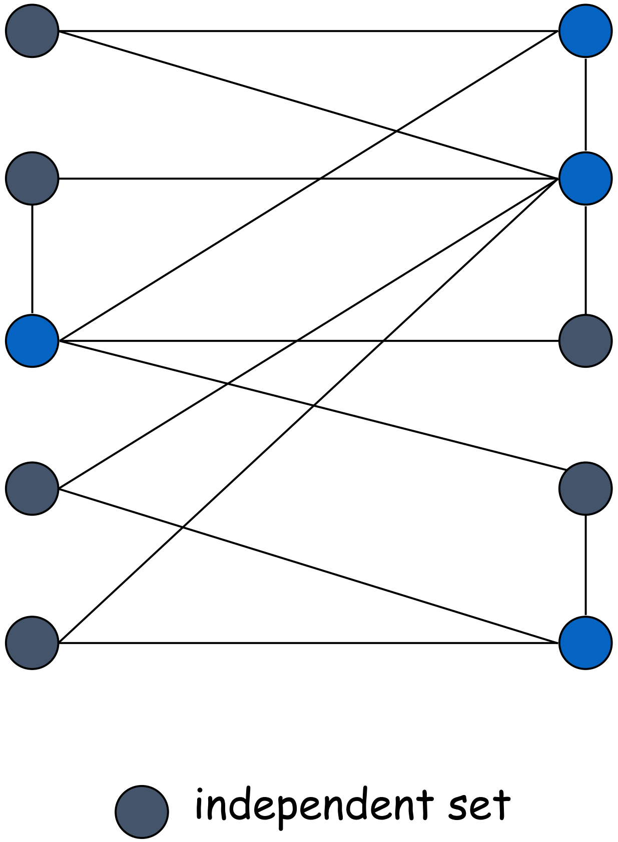
- Design algorithms. If $X \leq_P Y$ and Y can be solved in polynomial-time, then X can also be solved in polynomial time.

- Establish intractability. If $X \leq_P Y$ and X cannot be solved in polynomial-time, then Y cannot be solved in polynomial time.

- Establish equivalence. If $X \leq_P Y$ and $Y \leq_P X$, we use notation $X \equiv_P Y$.

Independent Set

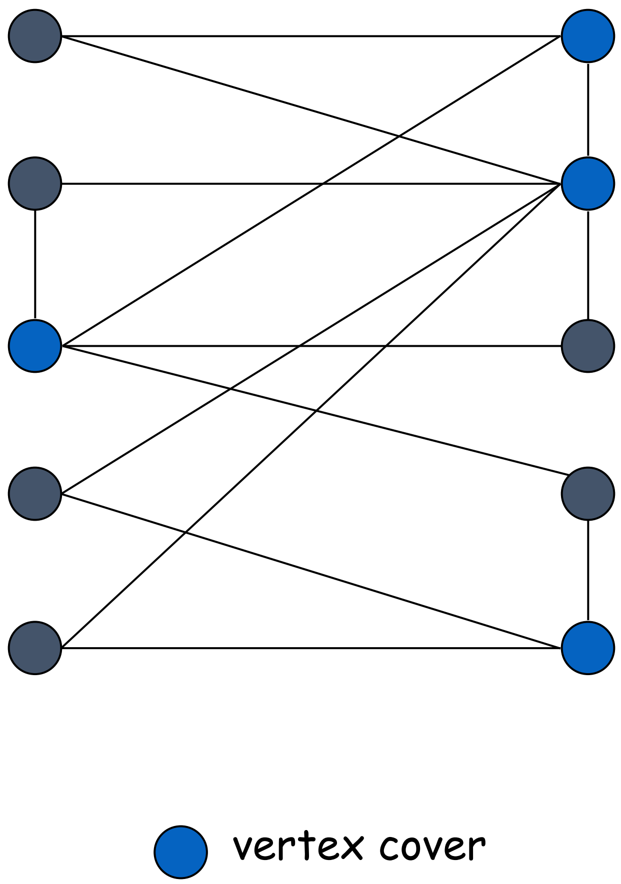
- INDEPENDENT SET: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?
- Ex. Is there an independent set of size ≥ 6? Yes.
- Ex. Is there an independent set of size ≥ 7? No.



Vertex Cover

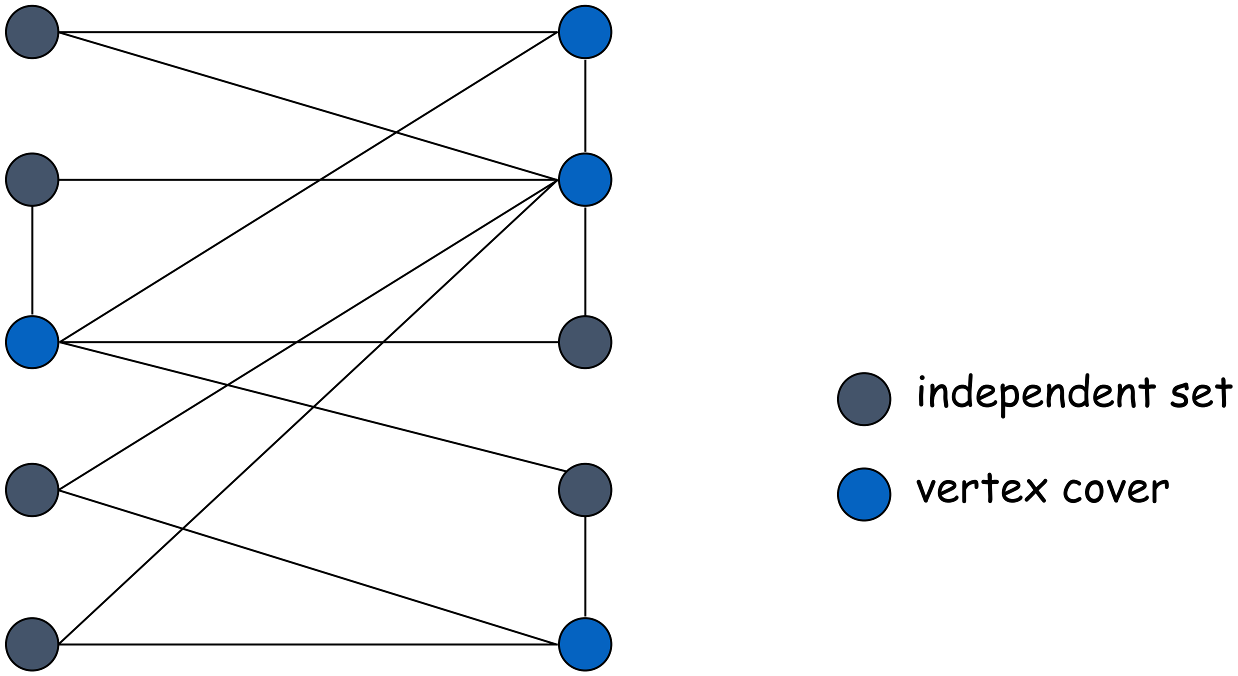
- VERTEX COVER: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in S?

- Ex. Is there a vertex cover of size ≤ 4? Yes.
- Ex. Is there a vertex cover of size ≤ 3? No.



Vertex Cover and Independent Set

- Claim. VERTEX-COVER $\equiv_P$ INDEPENDENT-SET.
- Pf. We show S is an independent set iff V – S is a vertex cover.



Vertex Cover and Independent Set

- Claim. VERTEX-COVER $\equiv_P$ INDEPENDENT-SET.
- Pf. We show S is an independent set iff V – S is a vertex cover.

- $\Rightarrow$
  - Let S be any independent set.
  - Consider an arbitrary edge (u, v).
  - S independent $\Rightarrow$ $u \notin S$ or $v \notin S$ $\Rightarrow$ $u \in V - S$ or $v \in V - S$.
  - Thus, V – S covers (u, v).

- $\Leftarrow$
  - Let V – S be any vertex cover.
  - Consider two nodes $u \in S$ and $v \in S$.
  - Observe that $(u, v) \notin E$ since V – S is a vertex cover.
  - Thus, no two nodes in S are joined by an edge $\Rightarrow$ S independent set. ▪

Set Cover

- SET COVER: Given a set U of elements, a collection S_1, S_2, . . . , S_m of subsets of U, and an integer k, does there exist a collection of $\leq$ k of these sets whose union is equal to U?

- Sample application.
  - m available pieces of software.
  - Set U of n capabilities that we would like our system to have.
  - The ith piece of software provides the set $S_i \subseteq U$ of capabilities.
  - Goal: achieve all n capabilities using fewest pieces of software.

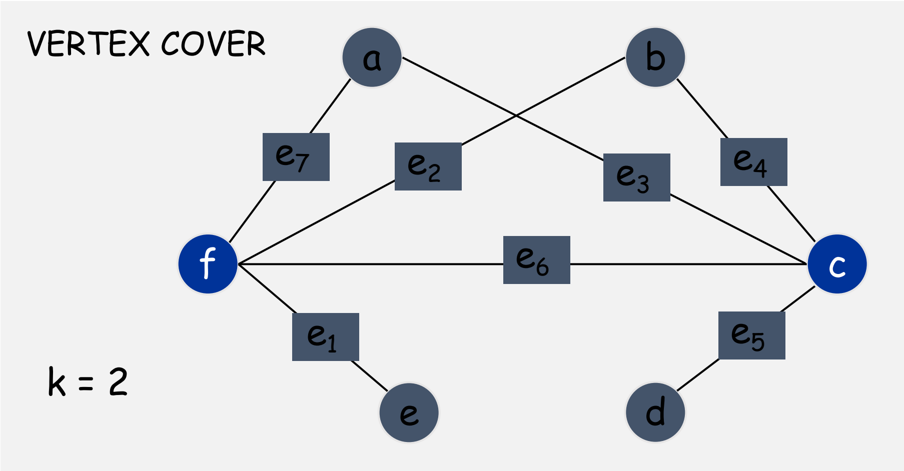
U = { 1, 2, 3, 4, 5, 6, 7 }

k = 2

S_1 = {3, 7}	S_4 = {2, 4}
S_2 = {3, 4, 5, 6}	S_5 = {5}
S_3 = {1}	S_6 = {1, 2, 6, 7}

Vertex Cover Reduces to Set Cover

- Claim. VERTEX-COVER $\leq_P$ SET-COVER.
- Pf. Given a VERTEX-COVER instance G = (V, E), k, we construct a set cover instance whose size equals the size of the vertex cover instance.



SET COVER

$U = \{1, 2, 3, 4, 5, 6, 7\}$
$k = 2$

$S_a = \{3, 7\}$ $S_b = \{2, 4\}$
$S_c = \{3, 4, 5, 6\}$ $S_d = \{5\}$
$S_e = \{1\}$ $S_f = \{1, 2, 6, 7\}$

Polynomial-Time Reduction

- Basic strategies.
  - Reduction by simple equivalence.
  - Reduction from special case to general case.
  - Reduction by encoding with gadgets.