CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

Structures can organize data in different types

- Declared using struct with member types and names included in braces.
- struct variables can be declared with the struct.

```
struct transaction
{
   int id;
   float amount;
   char name[20];
   char addr[30];
} t, *pt;
```

```
struct transaction
{
   int id;
   float amount;
   char name[20];
   char addr[30];
};
struct transaction
t, *pt;
```

```
typedef struct
{
   int id;
   float amount;
   char name[20];
   char addr[30];
} transaction;
transaction t,*pt;
```

Accessing struct members using . or ->

- A member in a **struct variable** can be access using **.**
- A member in a **struct pointed by a pointer** can be access using **->** or by dereferencing the pointer first and then using **.**

```
pt=&t;
printf("id: %d, name: %s, addr: %s",
        t.id, pt->name, (*pt).addr);
```

Pointer members in a struct

Some members need to have their memory dynamically allocated or location dynamically determined.

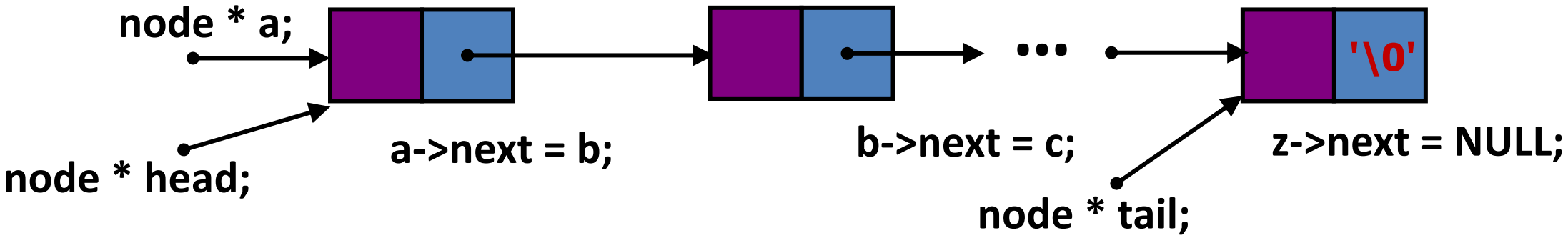
Extra pointers can be added to structs to support data structures, e.g., linked list, stack, queue, tree, graph, …

```
struct transaction{
   int id;
   float amount;
   char name[20];
   char *addr;
} t1, t2;
t1.addr=(char *)malloc(30);
t2.addr=another_str;
```

```
struct transaction{
   int id;
   float amount;
   char name[20];
   char *addr;
   struct transaction *next;
} t1, t2;
t1.next = &t2;
```

Linked Lists

- A linked list is a sequence of connected nodes.
- Each node contains at least
  - Some data
  - A pointer to the next node in the list
- The head pointer points to the first node
- The last node points to NULL
- The tail pointer (optional) points to the last node.

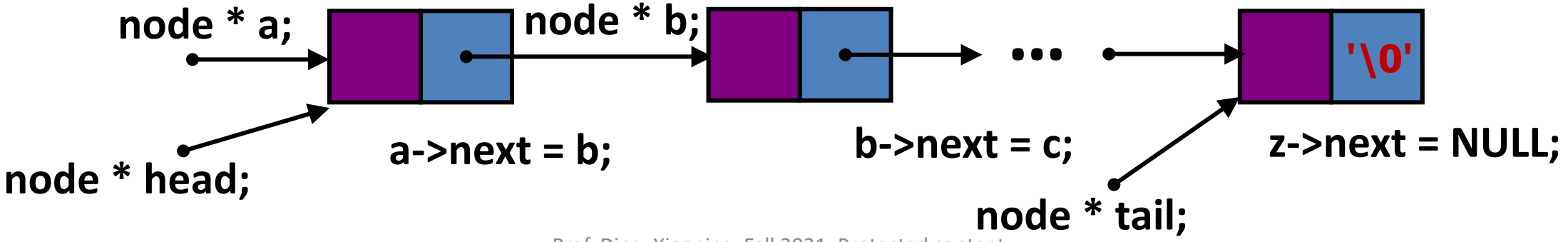


Why linked list?

- Often the maximum size of the list cannot be estimated.
  - Static arrays have fixed sizes.
  - Extending dynamic arrays (malloc-ed mem space) may need to copy data from the old and smaller space to a larger new space.
- Usually there are updates in the middle of the list, e.g., insertion, deletion, re-arranging, etc. Overhead is high with arrays.
  - Inserting a new element in the front or deleting the first element requires shifting all the elements in the array
  - On average, half of the lists needs to be moved for insertion/deletion.
- Compared to an array, linked list uses only as much space as is needed (requires extra-space for pointers)

Building a linked list

- **Declare node type** --- self-referential struct
- **Create nodes** --- allocate memory on-demand, initialize members
- **Link nodes to the list** --- find a location on the list (the previous and/or the next node) and update pointers in these nodes and the new node.
- **Keep the head pointer updated** --- If the address is lost, the whole list may be lost.
- **Ensure the next pointer of the last node to NULL**.
- **If there is a tail pointer, keep it updated**



```c
#include <stdio.h>
#include <stdlib.h>
struct node{
   int id;
   char name[20];
   struct node *next;
};

int main(){
   struct node *head=NULL, *tail=NULL, *pnode;
   while(1){
      pnode=(struct node *)malloc(sizeof(struct node));
      printf("id:"); scanf("%d", &(pnode->id));
      if(pnode->id<0) break;
      printf("name:"); scanf("%s", pnode->name);
      pnode->next=NULL; /*ensure next pointer of last node is NULL */
      if(head==NULL) head=pnode;
      if(tail!=NULL) tail->next=pnode;
      tail=pnode;
   }
```

Declare node type self_referential struct

head/tail pointers always point to the first/last node.

Create and initialize a new node

Link the new node to the end

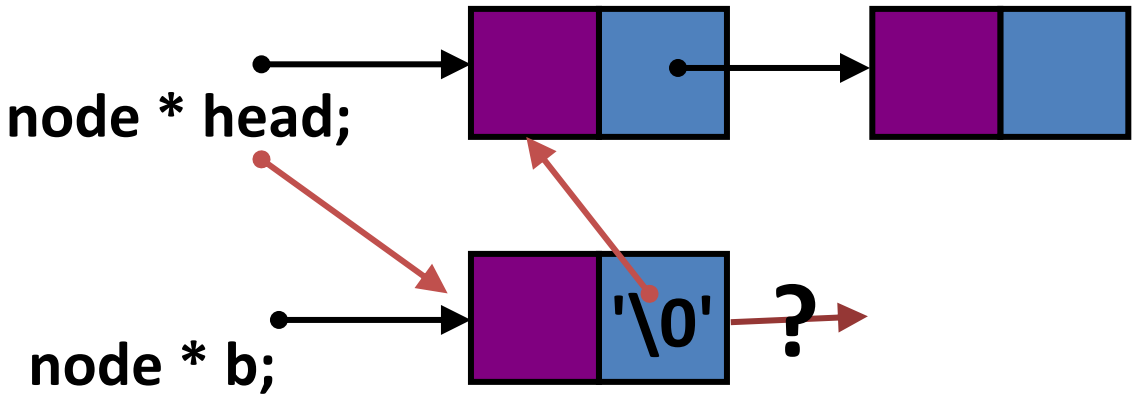
```
  pnode=head;
  while(pnode!=NULL) {
    printf("id: %d\t name:%s\n", pnode->id, pnode->name);
    pnode=pnode->next;
  }
}
```

Traverse a linked list

- Start from the head pointer.
- Proceed following the next pointers of nodes.
- Stop when the last node is reached.
  - Last node: next pointer is NULL, or pointed by the tail pointer.

Adding a node to a list

adding to the front



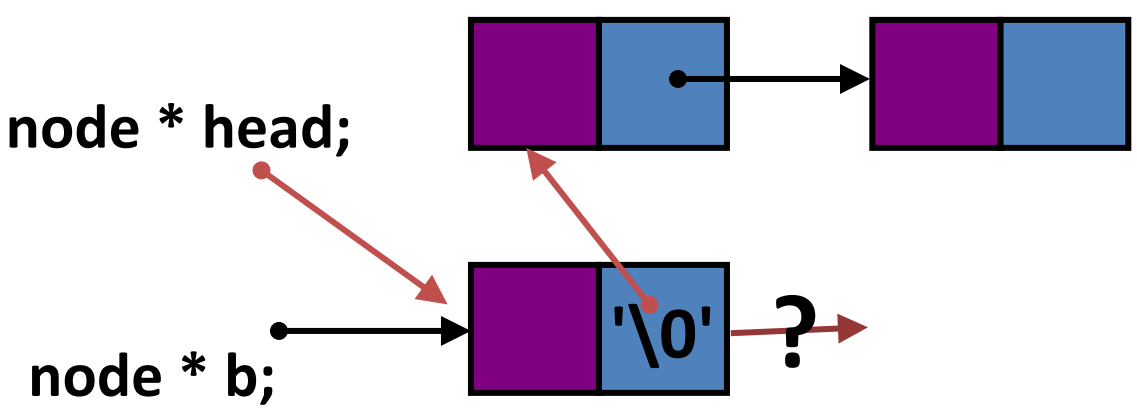

```
b->next = head;
head = b;
```

Order of the operations is important

```
head = b;
b->next = ?
```

Adding a node to a list

adding to the front

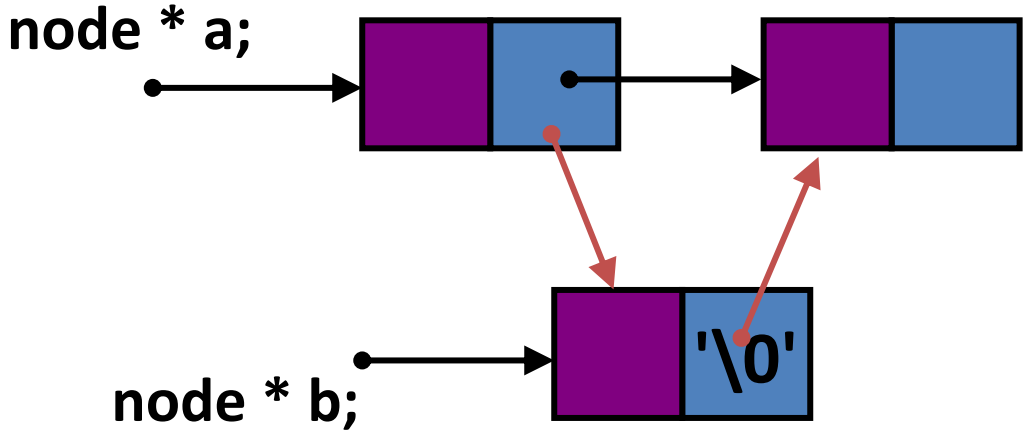



```
b->next = head;
head = b;
```

Order of the operations is important

```
head = b;
b->next = ?
```

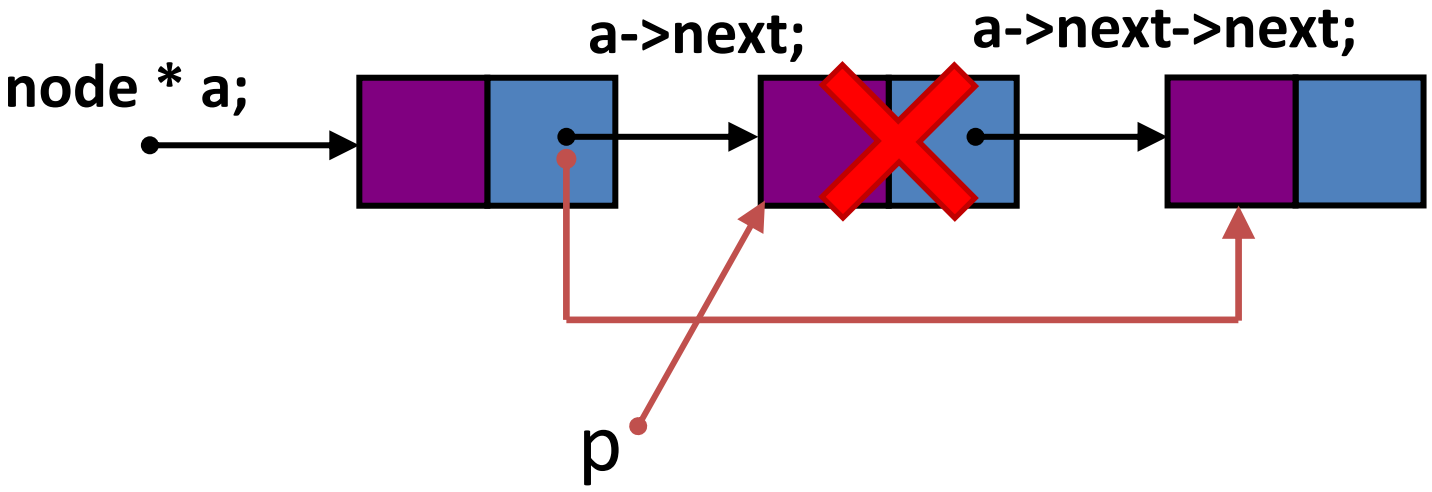
Inserting into the middle



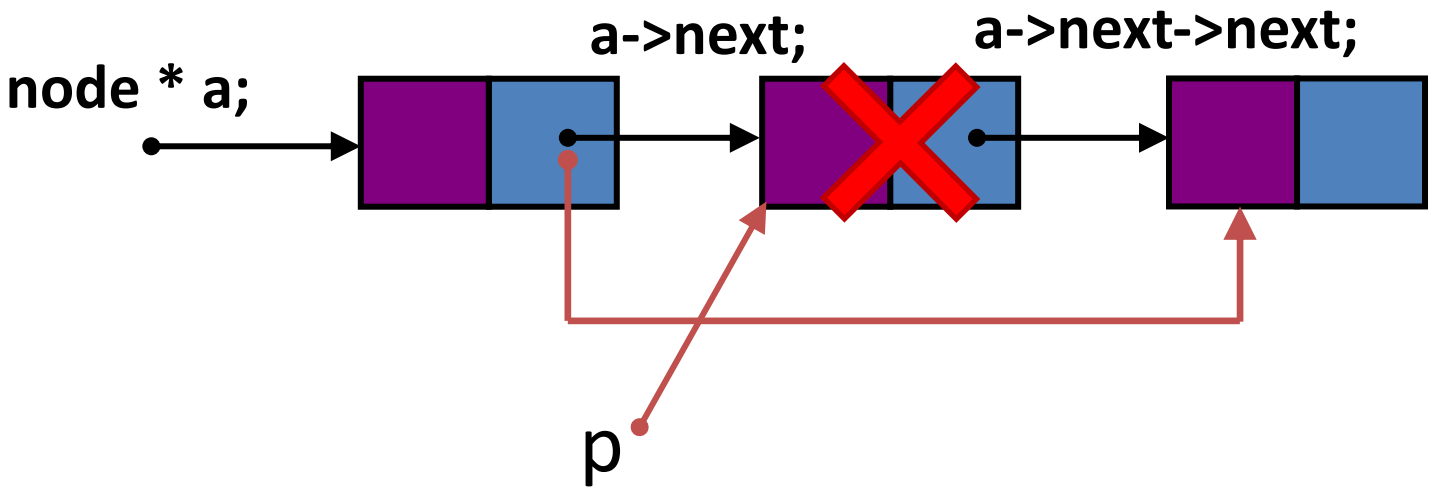

```
b->next = a->next;
a->next = b;
```

Consider a->next as a "head pointer" to the rest of the list.

Deleting a node




```
p = a->next;
a->next = a->next->next;
…
free(p);
```




```
p = a->next;
a->next = a->next->next;
…
free(p);
```

Deleting the first node

```
p = head;
head = head->next;
…
free(p);
```

Deleting the last node

?

Inserting/deleting a node: what to notice?

- Must handle different scenarios in different ways
  - E.g., for adding a node, check whether the new node is to be added to an empty list, to the front, the middle, or the end.
- Pay special attention to whether your code may dereference a NULL pointer.
  - Segmentation faults.
- Avoid losing the pointer pointing to a node
  - It becomes ``inaccessible'' if there are no pointers points to it.
- Keep the head pointer and the tail pointer (if you have one) updated.

Insertion sort

```
#include <stdio.h>
#include <stdlib.h>

struct node{
	int data;
	struct node *next;
};

void show_list(struct node *head) {
	struct node *p=head;
	if(head==NULL) return;
	while(p!=NULL){
		printf("->%d", p->data);
		p=p->next;
	}
	printf("\n");
}
```

```
struct node *add_to_sorted_list(
                                struct node *head,
                                struct node *p){
	struct node *prev_node, *curr_node;

	if(head==NULL) return p;
	if(p->data < head->data){
		p->next=head;
		return p;
	}
	prev_node=head;
	curr_node=prev_node->next;
	while(curr_node != NULL){
		if(p->data < curr_node->data) break;
		prev_node=curr_node;
		curr_node=curr_node->next;
	}
```

Insertion sort

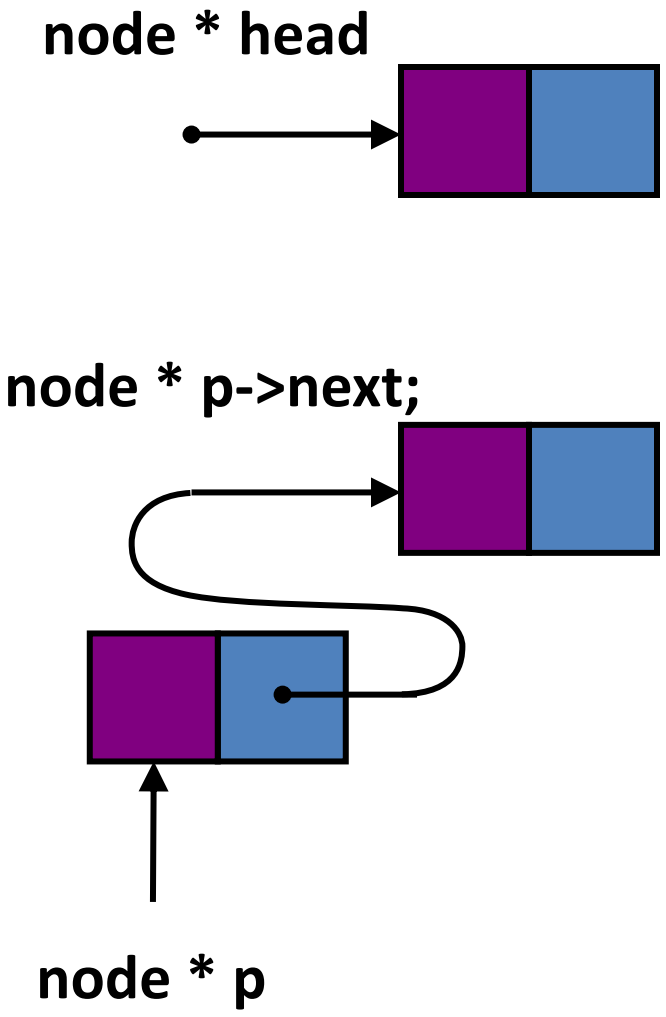
```
        prev_node->next=p;
        p->next=curr_node;
        return head;
}
int main(){
        int i;
        struct node *p, *head=NULL;
        while(scanf("%d",&i)!=EOF){
                p=(struct node *)malloc(sizeof(struct node));
                p->data=i;
                p->next=NULL;
                head=add_to_sorted_list(head,p);
        }
        show_list(head);
}
```

Using a dummy node to unify different scenarios

- With a dummy in the front, every node appear to be added after the next field of a node.
- Some linked list implementations keep the dummy node as a permanent part of the list.
  - Every list has a dummy node at its head.
  - The empty list is not represented by a NULL pointer.

```
struct node *dummy=(struct node *)
                    malloc(sizeof(struct node));
struct node *add_to_sorted_list(
                    struct node *head,
                    struct node *p){
    struct node *prev_node, *curr_node;
    dummy->next=head;
    head=dummy;
    prev_node=head;
    curr_node=prev_node->next;
    while(curr_node != NULL) {
        if(p->data < curr_node->data) break;
        prev_node=curr_node;
        curr_node=curr_node->next;
    }
    prev_node->next=p;
    p->next=curr_node;
    return dummy->next;
}
```

Using pointer to pointer to reduce # of scenarios




```
struct node* add_to_sorted_list(struct node *head,
                                struct node *p){
    struct node **curr, *next;

    if(head==NULL) return p;
    curr=&head;
    do{
        if(p->data < (*curr)->data) break;
        curr=&((*curr)->next);
    }while(*curr!=NULL);
    next=*curr;
    *curr=p;
    p->next=next;
    return head;
}
```

Debugging linked list programs

Segmentation fault (segfault)

- caused by a program accessing the memory it should not
- Use after free, Dereferencing NULL pointers
- Use gdb
  - Run program in gdb
  - Execution stop on segfault
  - Check stack using backtrace (bt) or where command.
  - check the values of related variables and pointers.

```
struct node *add_to_sorted_list(
        struct node *head,  struct node *p){
    struct node *prev_node, *curr_node;
    if(head==NULL) return p;
    if(p->data < head->data){
        p->next=head;
        return p;
    }
    prev_node=head;
    curr_node=prev_node->next;
    while (p->data < curr_node->data){
        prev_node=curr_node;
        curr_node=curr_node->next;
    }
    prev_node->next=p;
    p->next=curr_node;
    return head;
}
```

Debugging linked list programs

Show linked list every time after the list is changed.

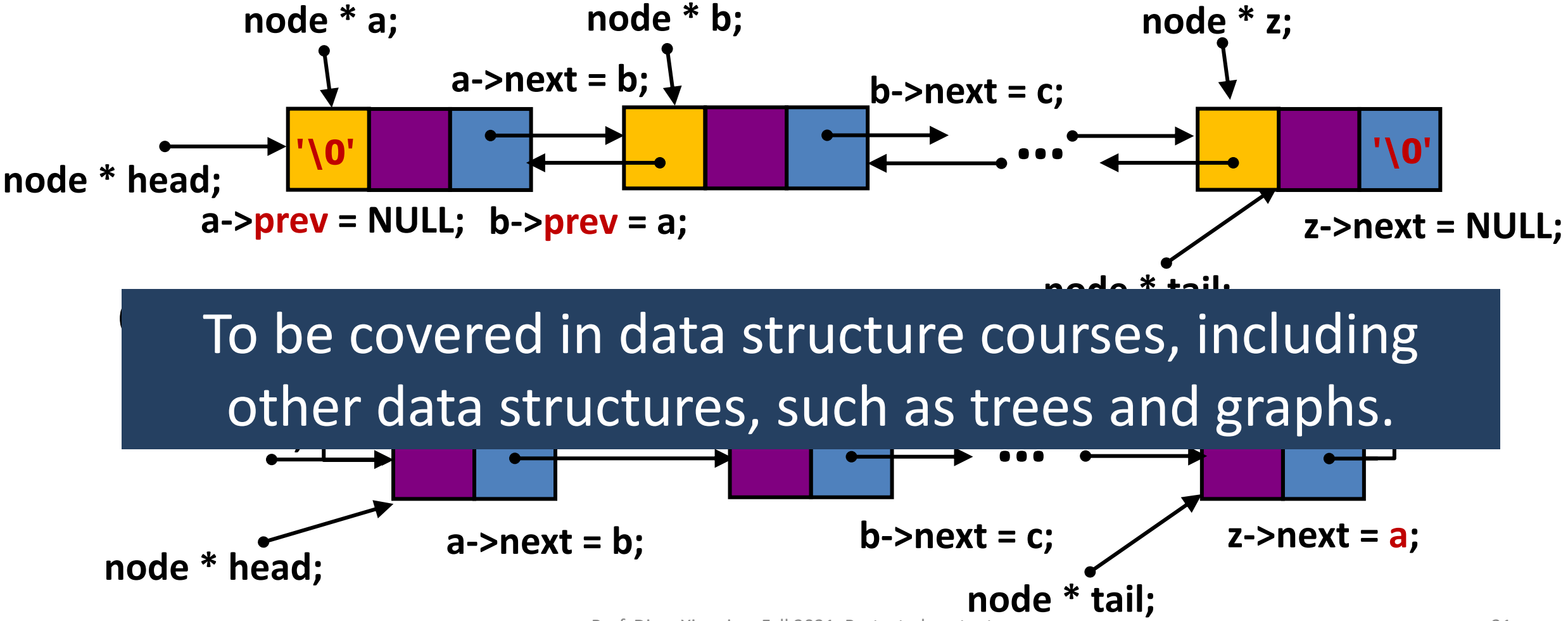
```
1
->1
3
->1->3
5
->1->5->3
7
->1->7->5->3
```

```
struct node *add_to_sorted_list(
        struct node *head, struct node *p){
    struct node *prev_node, *curr_node;

    if(head==NULL) return p;
    if(p->data < head->data){
        p->next=head;
        return p;
    }

    prev=head;
    curr_node=prev_node->next;
    while(curr_node != NULL) {
        if(curr_node->data >= < p->data)
                    break;
        prev_node=curr_node;
        curr_node=curr_node->next;
    }
    prev_node->next=p;
    p->next=curr_node;
    return head;
}
int main(){
    int i;
    struct node *p, *head=NULL;
    while(scanf("%d",&i)!=EOF){
        p=(struct node *)malloc(
                sizeof(struct node));
        p->data=i;
        p->next=NULL;
        head=add_to_sorted_list(head,p);
        show_linkedlist(head);
    }
    show_linkedlist(head);
}
```

Other types of linked list

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

Double linked list



CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

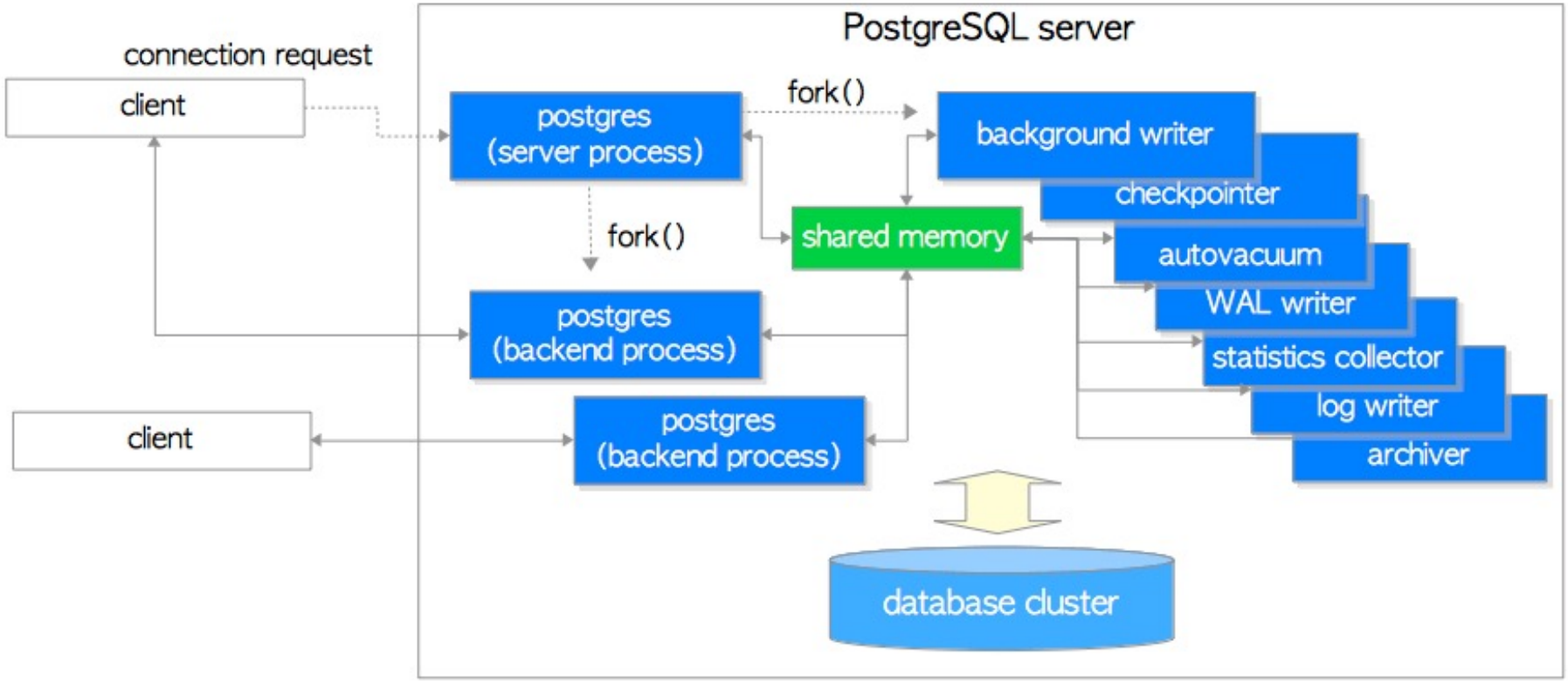
Programs, processes, and threads

- Program: source code or executable file
- Process: running program
  - A binary image loaded into memory
  - An instance of virtual memory
    - Heap and stacks
  - kernel resources and data structures
    - Records about which files have been opened, memory that has been allocated, etc.
    - a unique process ID number(PID)
  - One or more threads
- Thread: a unit of activity inside of a process, corresponding to a virtualized processor
  - a stack
  - processor state such as registers and an instruction pointer
  - Threads in a process share the same virtual memory space

The use of multiple threads, multiple processes, and multiple programs

- The use of multiple threads (multi-threading) and/or multiple processes
  - Objectives
    - To enhance parallel processing
    - To increase response to the user
    - To utilize the idle time of the CPU (e.g., scheduling another thread/process when one thread/process is waiting for I/O)
  - Multiple threads or multiple processes are created in the same program.
  - Threads vs. processes: a trade-off between isolation and data sharing
- The use of multiple programs
  - Each program is a process.
  - To fit the need of running multiple tools and different tools developed in different programs/developers.
  - Most common case: shell

Postgresql Internals



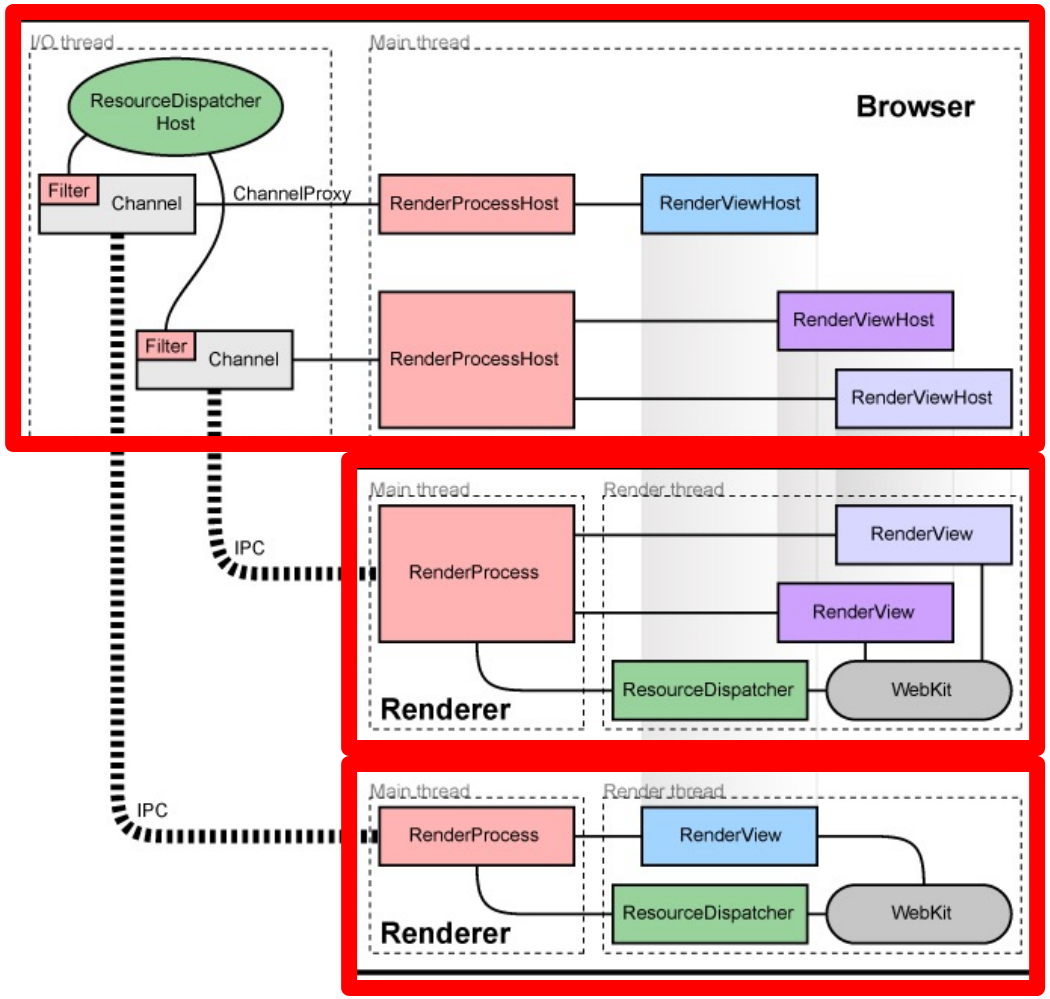
- A **postgres server process** is a parent of all processes related to a database cluster management.
- Each **backend process** handles all queries and statements issued by a connected client.
- Various **background processes** perform processes of each feature (e.g., VACUUM and CHECKPOINT processes) for database management.

http://www.interdb.jp/pg/pgsql02.html

MySQL implements concurrent connections by spawning a thread-per-connection.

- relatively low overhead;
  - spawning a new thread occupies less memory than forking a new process in term of both time and memory.
  - Synchronization and data exchange incur much less cost between threads than between processes.
  - Context switches is less costly between threads than between processes
- It’s not uncommon to scale MySQL to 10,000 or so concurrent connections.
- significant problems when scaling PostgreSQL past a few hundred active connect.

Google Chrome architecture



- Separate processes for browser tabs to protect the overall application from bugs and glitches in the rendering engine
- Restricted access from each rendering engine process to others and to the rest of the system

https://sites.google.com/a/chromium.org/dev/developers/design-documents/multi-process-architecture

POSIX thread: pthread

Pthreads --- POSIX thread library.

- POSIX (Portable Operating System Interface)
  - an IEEE standard that sets how UNIX systems look, act, and feel.
- The POSIX threads API is available on almost all UNIX-like operating systems.
  - if you write parallel code using Pthreads on a Linux machine, it will likely work on other UNIX variants.
- For C programs on Linux, pthreads API is implemented in Native POSIX Thread Library (NPTL).
- Need to link pthread library when compiling the source code.

gcc mypthreadprog.c -pthread -o mypthreadprog

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* The "thread function" passed to pthread_create.  Each thread
 * executes this function and terminates when it returns from
 * this function. */
void *HelloWorld(void *id) {
    /* We know the argument is a pointer to a long, so we cast it
     * from a generic (void *) to a (long *). */
    long *myid = (long *) id;

    printf("Hello world! I am thread %ld\n", *myid);

    return NULL; // We don't need our threads to return anything.
}

int main(int argc, char **argv) {
    int i;
    int nthreads; //number of threads
    pthread_t *thread_array; //pointer to future thread array
    long *thread_ids;
```

```
    // Read the number of threads to create from the command line.
    if (argc !=2) {
        fprintf(stderr, "usage: %s <n>\n", argv[0]);
        fprintf(stderr, "where <n> is the number of threads\n");
        return 1;
    }
    nthreads = strtol(argv[1], NULL, 10);

    // Allocate space for thread structs and identifiers.
    thread_array = malloc(nthreads * sizeof(pthread_t));
    thread_ids = malloc(nthreads * sizeof(long));

    // Assign each thread an ID and create all the threads.
    for (i = 0; i < nthreads; i++) {
        thread_ids[i] = i;
        pthread_create(&thread_array[i], NULL, HelloWorld, &thread_ids[i]);
    }

    /* Join all the threads. Main will pause in this loop until all threads
     * have returned from the thread function. */
    for (i = 0; i < nthreads; i++)  pthread_join(thread_array[i], NULL);

    free(thread_array);     free(thread_ids);     return 0;
}
```

Creating and using worker threads

- Include header file **pthread.h**

- Implement a **thread function**.
  - analogous to a main() function for a worker (created) thread. — a thread begins execution at the start of its thread function and terminates when it reaches the end.
  - Each thread executes the thread function using its private execution state, including its own stack and registers.
  - Input and output: **void *** --- Addresses where real input data and output data are saved.

- Main thread (main() function) creates and “joins” worker threads.
  - The program starts as a single-threaded process --- main thread
  - Create worker threads: **pthread_create()** function
  - Join worker threads: **pthread_join()** function

Creating a thread - pthread_create()

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
   void * thread_function, void * thread_args);
```

- pthread_t ***thread**: address of a thread struct --- a handle to the thread.
- **attr**: thread attributes. Usually set to NULL to use default thread attributes.
- **thread_function**: name of the function the thread should execute.
- **thread_args**: where are the arguments to pass to thread function when it starts
  - Cannot share arguments between threads (data race, one thread modifies while others read).
- pthread_create() **returns 0 if succeed**, and a nonzero error code otherwise.

```
for (i = 0; i < nthreads; i++) {
  thread_ids[i] = i;
  pthread_create(&thread_array[i], NULL, HelloWorld, &thread_ids[i]);
}
```

Joining thread

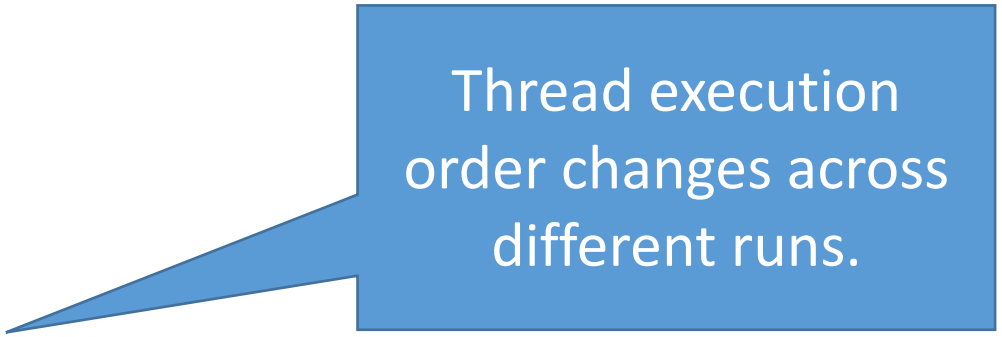
```
int pthread_join (pthread_t thread, void **retval);
```

- pthread_join() function suspends the execution of its caller until the thread its references terminates.
  - Called by the main thread to wait for the finish of worker threads.
  - When the main thread terminates, all other threads are terminated too.
- **thread**: which thread to wait on
- **retval**: where the thread's return value should be stored
  - *retval is a pointer returned by the thread function.

```
for (i = 0; i < nthreads; i++)
    pthread_join(thread_array[i], NULL);
```

Compile and run the program

```
$ gcc -o hellothreads hellothreads.c -pthread
$ ./hellothreads
usage: ./hellothreads <n>
where <n> is the number of threads
$ ./hellothreads 4
Hello world! I am thread 1
Hello world! I am thread 2
Hello world! I am thread 3
Hello world! I am thread 0
$ ./hellothreads 4
Hello world! I am thread 0
Hello world! I am thread 1
Hello world! I am thread 2
Hello world! I am thread 3
```



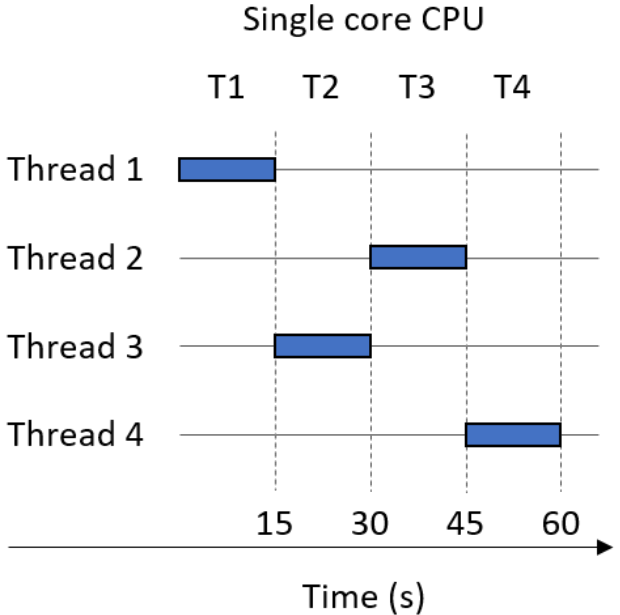
Another example: scalar multiplication

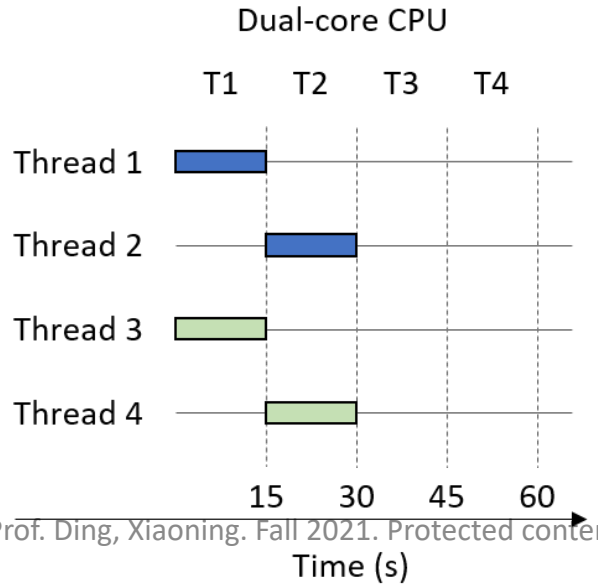
Sequential code of multiplying an array and an integer

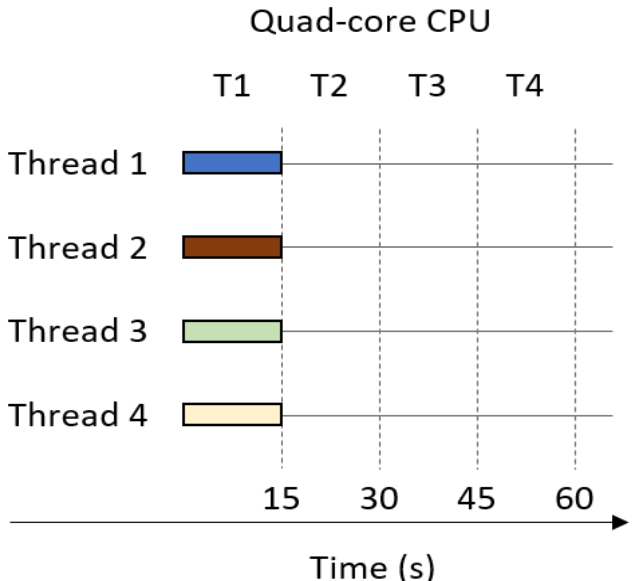
```
void scalar_multiply(int * array, long length, int s) {
   for (i = 0; i < length; i++)
      array[i] = array[i] * s;
}
```

To create a multi-threaded version of this application with t threads, it is necessary to:

- Create *t* threads.
- Assign each thread a subset of the input array (i.e. *length/t* elements).
- Instruct each thread to multiply the elements in its array subset by *s*.







```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

long *array, length, nthreads, s;

void *scalar_multiply(void *id) {
    long *myid = (long *) id;
    int i;

    //assign each thread its own chunk of elements to process
    long chunk = length / nthreads;
    long start = (*myid) * chunk;
    long end  = start + chunk;

    if ( (*myid) == nthreads - 1) end = length;

    //perform scalar multiplication on assigned chunk
    for (i = start; i < end; i++) {
        array[i] *= s;
    }

    return NULL;
}
```

```
int main(int argc, char **argv) {
    int i;
    pthread_t *thread_array; //pointer to future thread array
    long *thread_ids;

    nthreads = strtol(argv[1], NULL, 10);
    length = strtol(argv[2], NULL, 10);
    s = strtol(argv[3], NULL, 10);
    array = (long *)malloc(sizeof(long)*length);
    thread_array = malloc(nthreads * sizeof(pthread_t));
    thread_ids = malloc(nthreads * sizeof(long));
    ...  /* initialize array */
    for (i = 0; i < nthreads; i++) {
        thread_ids[i] = i;
        pthread_create(&thread_array[i], NULL,
                       scalar_multiply, &thread_ids[i]);
    }

    for (i = 0; i < nthreads; i++)  pthread_join(thread_array[i], NULL);
    ...  /* print array or save it into file */
    free(thread_array);  free(thread_ids); free(array);  return 0;
}
```

Private data and shared data

- Global variables are shared by all threads.
- Local variables declared in thread function (stack variables) are private to each thread.
- Data shared between main thread and a worker thread
  - Input arguments of thread function passed via a pointer.
  - Data returned by thread function passed via another pointer.

Avoid using global variables by passing data to threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

struct t_arg {
    int *array; // pointer to shared array
    long length; // num elements in array
    long s; //scaling factor
    long numthreads; // total number of threads
    long id; //  logical thread id
};
void * scalar_multiply(void* args) {
    //cast to a struct t_arg from void*
    struct t_arg * myargs = (struct t_arg *) args;

    //extract all variables from struct
    long myid =  myargs->id;
    long length = myargs->length;
    long s = myargs->s;
    long nthreads = myargs->numthreads;
    int * ap = myargs->array; //pointer to array in main
```

```
 //code as before
    long chunk = length/nthreads;
    long start = myid * chunk;
    long end  = start + chunk;
    if (myid == nthreads-1)  end = length;

    for (int i = start; i < end; i++)  ap[i] *= s;
    return NULL;
}

int main(int argc, char **argv) {
    int i;
    pthread_t *thread_array; //pointer to future thread array
    long *thread_ids;
    long nthreads = strtol(argv[1], NULL, 10); //get number of threads
    long length = strtol(argv[2], NULL, 10); //get length of array
    long s = strtol( argv[3], NULL, 10 ); //get scaling factor
    int *array = malloc(length*sizeof(long));
    pthread_t *thread_array = malloc(nthreads * sizeof(pthread_t));
    struct t_arg *thread_args = malloc(nthreads * sizeof(struct t_arg));
```

```
    //Populate thread arguments for all the threads
    for (i = 0; i < nthreads; i++){
        thread_args[i].array = array;
        thread_args[i].length = length;
        thread_args[i].s = s;
        thread_args[i].numthreads = nthreads;
        thread_args[i].id = i;
    }

    ...  /* initialize array */
    for (i = 0; i < nthreads; i++)
        pthread_create(&thread_array[i], NULL,
                       scalar_multiply, &thread_args[i]);


    for (i = 0; i < nthreads; i++)  pthread_join(thread_array[i], NULL);
    ...  /* print array or save it into file */
    free(thread_array);  free(thread_ids); free(array);  return 0;
}
```

common “bugs” that first-time pthread programmers make

- When creating multiple threads, pthread_t struct and/or thread argument are reused across threads and overwritten.
- pthread_join() is not called in main thread
  - the main thread may reach the end of main() and exit
  - other threads are terminated when main thread finishes
    - It possible that they have NOT had a chance to compute a result
- The following code is sequential!

```
for (i=0; i < num_threads; i++) {
    pthread_create(&(threads[i]),...)
    pthread_join(threads[i],...)
}
```

Synchronizing Threads

Why is synchronization important?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>
#define NUM_THREADS 2
#define ITERATIONS_PER_THREAD 5000000
int cnt = 0;
void * worker( void *ptr )
{
        int i;
        for (i = 0; i < ITERATIONS_PER_THREAD; i++)
                cnt++;
}

int main(void)
{
        pthread_t threads[NUM_THREADS];
        int i, result;
        /* Start threads */
```

```
    for (i = 0; i < NUM_THREADS; i++) {
            result = pthread_create(&threads[i], NULL,
                                    worker, NULL);
            assert(result == 0);
    }
    /* Wait for threads to finish */
    for (i = 0; i < NUM_THREADS; i++) {
            result = pthread_join(threads[i], NULL);
            assert(result == 0);
    }
    printf("Final value: %d (%.2f%%)\n", cnt,
                100.0 * cnt / (NUM_THREADS *
                               (double)ITERATIONS_PER_THREAD));
}
```

What will be printed out if everything works as expected?

What will be printed out actually?

How small/large can the final value be?

Shared memory “code” for increasing a counter

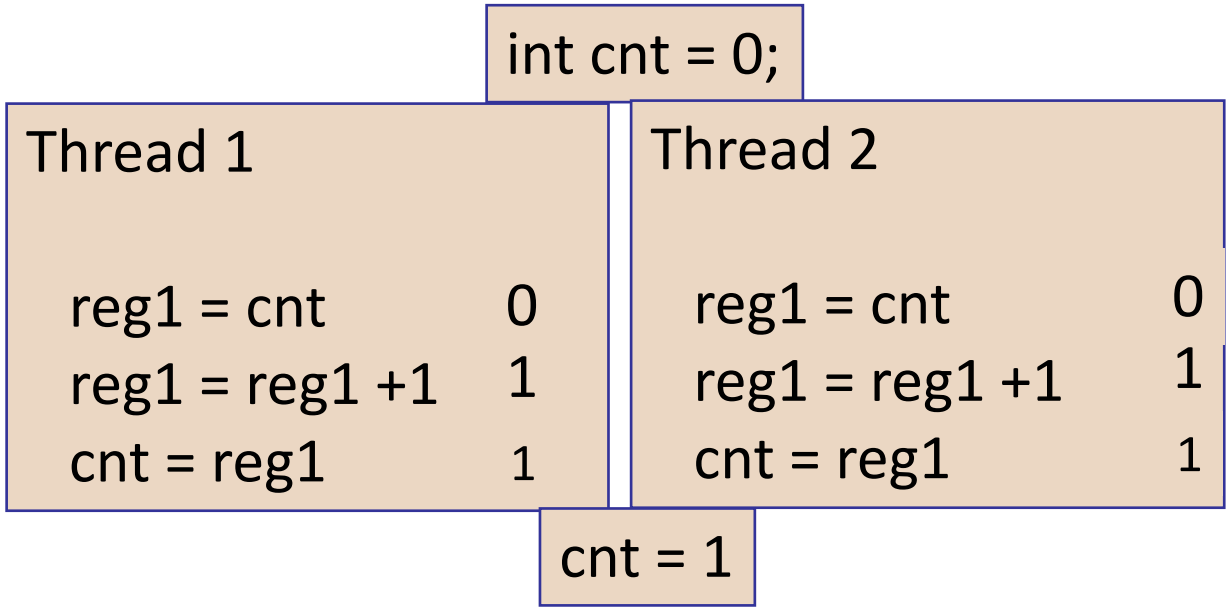
```
int cnt = 0;
```

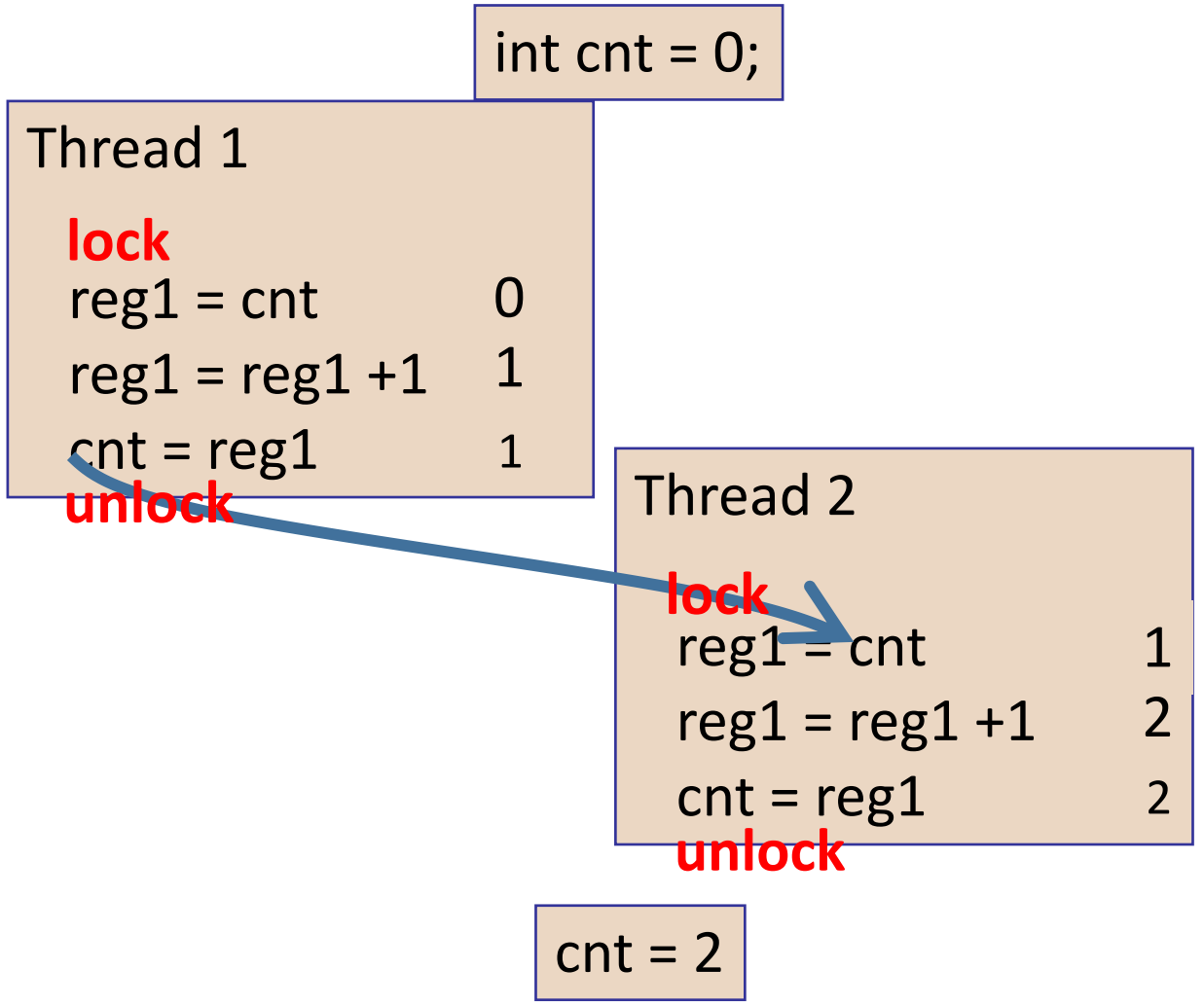
```
Thread 1

  for (i = 0, i<100; i++)
        cnt++;
```

```
Thread 2

  for (i = 0, i<100; i++)
        cnt++;
```

- Problem is a race condition on the shared variable ***cnt*** in the program
- A race condition or data race occurs when:
  - two threads share the same variable, and at least one does a write.
  - The accesses are not synchronized so they could happen simultaneously

Shared memory “code” for increasing a counter





Semaphores

- A non-negative global integer synchronization variable
- Manipulated by wait and post operations:
  - wait(s): [while (s == 0) wait(); s--;]
    - Also P(s), Dutch for "Proberen" (test)
  - post(s): [s++;]
    - Also V(s), Dutch for "Verhogen" (increment)
- OS kernel guarantees that operations between brackets [ ] are executed indivisibly
  - i.e., s-- can’t be broken into load/update/store
  - Result: only one wait or post operation at a time can modify s
  - When while-loop in wait terminates, only that wait can decrement s
  - S never goes below 0

Semaphores (unnamed)

- Include header file `semaphore.h`
- Declare semaphore as global variable and thus shared between threads
  - `sem_t  mysem;`
- Initialize the semaphore using **sem_init()** (usually in main()).
  - `int sem_init(sem_t *sem,int pshare, unsigned int value);`
  - sem: pointer to semaphore
  - pshare: 0 --- shared between threads; 1 --- shared between process
  - value: initial value of semaphore
- Destroyed using **sem_destroy()**
  - `int sem_destroy(sem_t *sem);`
- Wait operation using **sem_wait()**, and Post operation using **sem_post()**
  - `int sem_wait(sem_t *s);`
  - `int sem_post(sem_t *s);`

```
#include <semaphore.h>
...
int cnt = 0;
sem_t cnt_sem;
void * worker( void *ptr ){
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++) {
        sem_wait(&cnt_sem);
        cnt++;
        sem_post(&cnt_sem);
    }
}
int main(void){
  ...
  result = sem_init(&cnt_sem, 0, 1); /* Initialize semaphore */
  if (result < 0)
     exit(-1);
  ...
  sem_destroy(&cnt_sem);
}
```

Mutexes in pthreads

- If possible, use mutexes over semaphores
  - Mutexes: only locking thread can unlock
  - Semaphores: any thread can decrement (unlock); much harder to manage
- To declare a mutex: `pthread_mutex_t  amutex;`
- To initialize a mutex:

```
amutex = PTHREAD_MUTEX_INITIALIZER;
```
or
```
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(&amutex);
int pthread_mutex_unlock(&amutex);
```

- To destory a mutex

```
int pthread_mutex_destroy(&amutex);
```

Barrier -- global synchronization

- A point in a program where all threads must reach before any thread can cross
  - threads reach the barrier & then wait until all other threads arrive
  - all threads reach & begin executing code beyond the barrier
- simple use of barriers -- all threads hit the same one
- for(…){

```
work_on_my_problem();
barrier;
get_data_from_others();
barrier;
}
```

- more complicated -- barriers on branches (or loops)

```
if (tid % 2 == 0) {
    work1();
    barrier
}
else {
    barrier
}
```

Creating and Initializing a Barrier

- To declare a barrier:

```
pthread_barrier_t b;
```

- To initialize a barrier for 3 threads:

```
pthread_barrier_init(&b,NULL,3); or
b=PTHREAD_BARRIER_INITIALIZER(3);
```

- To wait at a barrier:

```
pthread_barrier_wait(&b);
```

- To destroy a barrier

```
pthread_barrier_destroy(&b);
```

A more complicated example (multi-threaded CountSort)

CountSort

- Designed for sorting many integer values within a small range.
- For an array, CountSort counts the frequency of each value in the array, and then enumerates each value by its frequency.

```
void countElems(int *counts,
 int *array_A,long length) {

 int val, i;
 for(i = 0; i < length; i++){
   val=array_A[i];
   counts[val]=counts[val]+1;
 }
}
```

A = [9, 0, 2, 7, 9, 0, 1, 4, 2, 2, 4, 5, 0, 9, 1]

Count the frequency of each value from 0 to 9

counts = [3, 2, 3, 0, 2, 1, 0, 1, 0, 3]

2 1s

3 0s

Emulate each value by its frequency

A = [0, 0, 0, 1, 1, 2, 2, 2, 4, 4, 5, 7, 9, 9, 9]

How to parallelize countElems?

https://diveintosystems.org/singlepage/book/modules/SharedMemory/assets/attachments/countElems_p_v2.c

```
pthread_mutex_t mutex;
void *countElems( void *args ) {
    struct t_arg * myargs = (struct t_arg *)args;
    int *array = myargs->ap;
    long *counts = myargs->countp;

    long chunk = length / nthreads, start = myid * chunk;
    long end = (myid + 1) * chunk, val, i;
    if (myid == nthreads - 1) end = length;
    pthread_mutex_lock(&mutex);
    for (i = start; i < end; i++) {
        val = array[i];
        counts[val] = counts[val] + 1;
    }
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

https://diveintosystems.org/singlepage/book/modules/SharedMemory/assets/attachments/countElems_p_v3.c

Computation is still serial.

No speed up.

Each thread accumulate frequencies into a local array and then accumulate local frequencies to global array

https://diveintosystems.org/singlepage/book/modules/SharedMemory/assets/attachments/countElems_p_v3.c

```
void *countElems( void *args ) {
    struct t_arg * myargs = (struct t_arg *)args;
    int *array = myargs->ap, *counts = myargs->countp;
    long local_counts[MAX] = {0};

    long chunk = length / nthreads, start = myid * chunk;
    long end = (myid + 1) * chunk, val, i;
    if (myid == nthreads-1) end = length;

    for (i = start; i < end; i++) {
        val = array[i];
        local_counts[val] = local_counts[val] + 1;
    }
    pthread_mutex_lock(&mutex);
    for (i = 0; i < MAX; i++) counts[i] += local_counts[i];
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

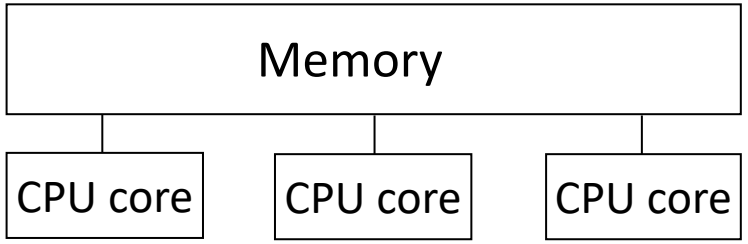
Most computation can be done in parallel.

CS 288 Intensive Programming in Linux

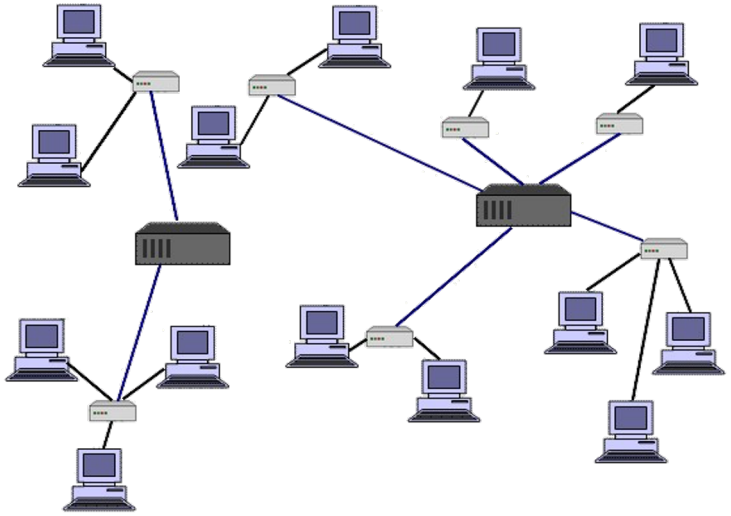
Professor Ding, Xiaoning

Two architectures and two programming models

Shared Memory System



Distributed Memory System



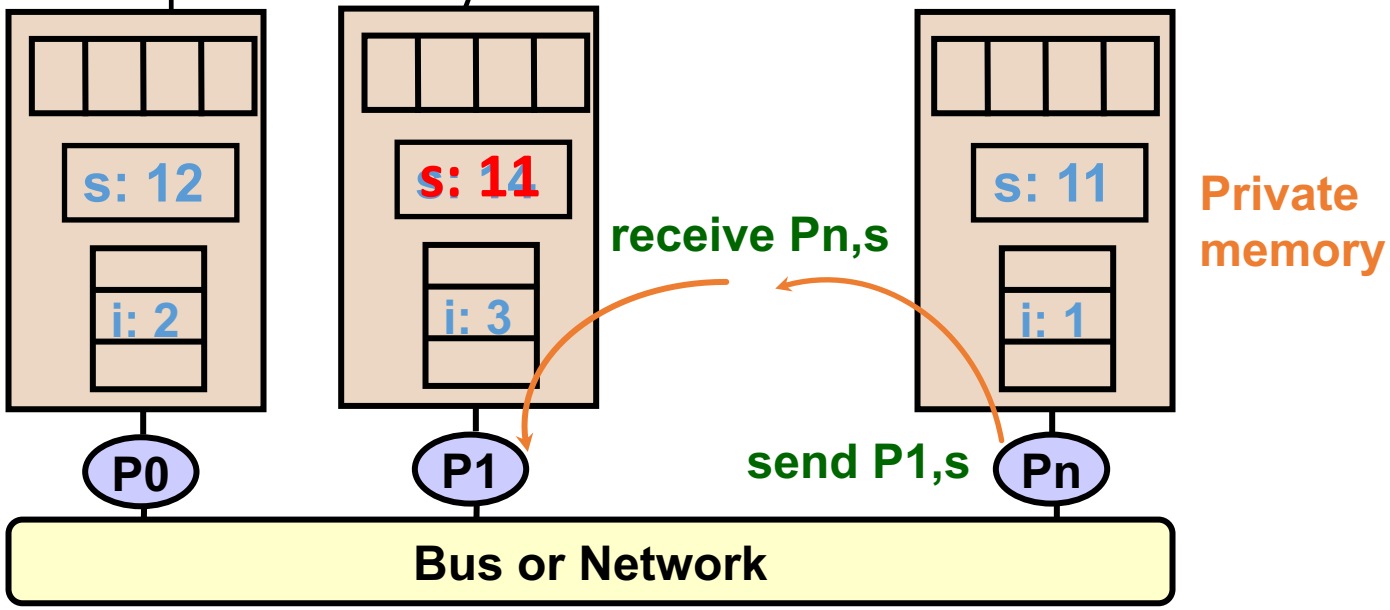
- Shared memory programming
  - pthread, open MP
  - Usually used on shared memory systems
- Message passing programming
  - processes, open MPI
  - Can be used on both types of systems
- Using multiple computers to increase computing capacity and speed
  - More sources: CPU cores, memory capacity, storage bandwidth, etc.
  - Many scenarios need super computing, e.g., fluid dynamics simulation, AI training, etc.

What is Message Passing Interface(MPI)

- A standard interface for implementing message-passing libraries:
  - Extended message-passing model
  - Not a language or compiler specification, not a specific implementation or product
  - 125 functions, but only 6 basic functions.
  - Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required
- Software packages implementing the interface and the related tools.
  - Usually open-source, mainly includes library for message passing, compiler wrapper, and supporting tools.
  - Dominant MPI implementations on Linux: Open MPI and MPICH
  - Other implementations: Microsoft MPI
- Hide the complexity and difference of parallel computing architectures
  - Develop one program, run on different systems --- super computers, cluster of heterogeneous servers, etc.

MPI program

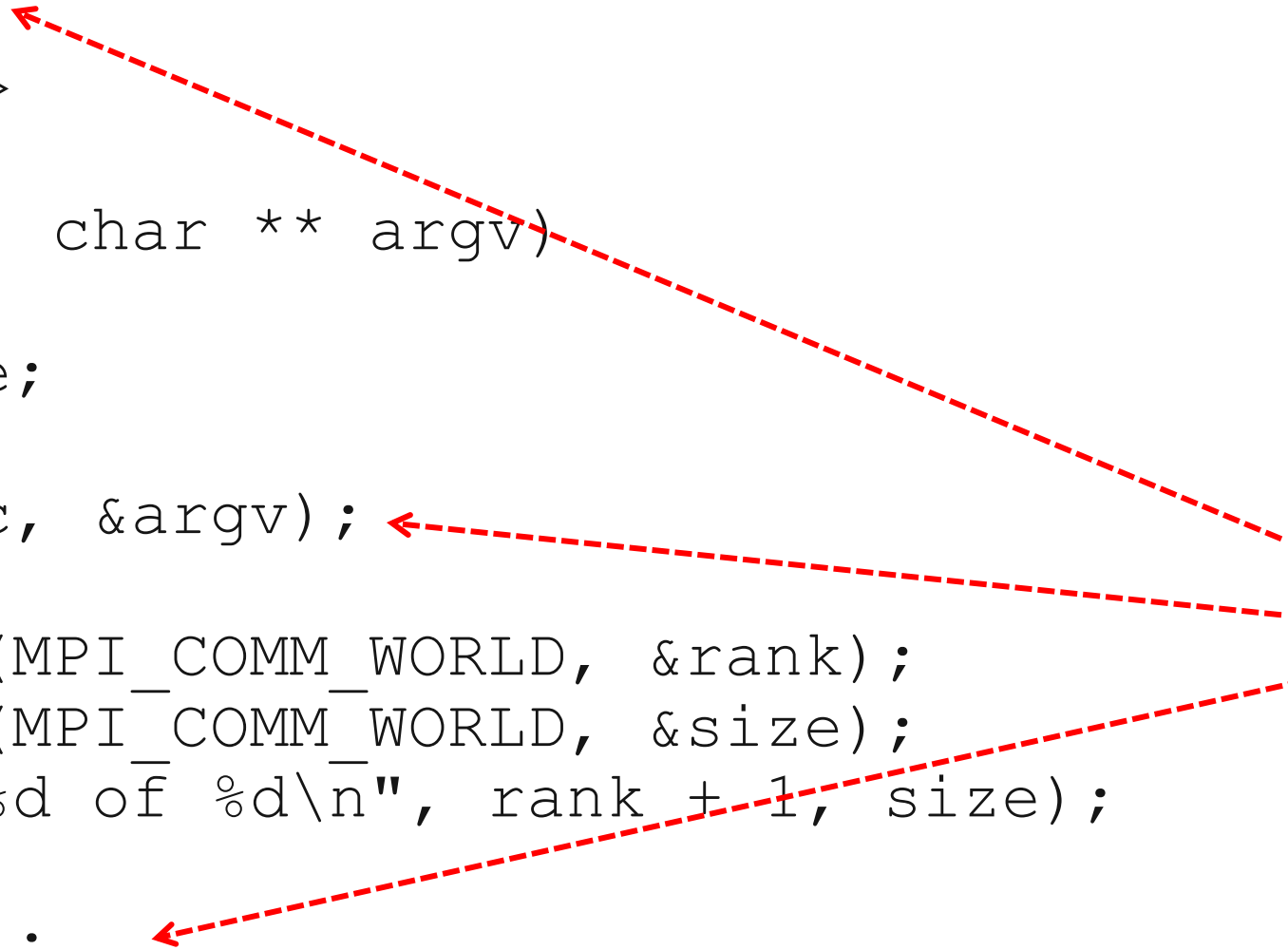
- A program consists of a collection of processes.
  - Usually created at program startup time and kept fixed during execution
- SPMD (Single Program Multi-Data)
  - Multiple processes execute the same copy of code
  - Each process is a thread of control with its own local address space.
- NO shared data between processes.
  - Processes communicate by explicit `send` and `receive` function pairs
  - Coordination is implicit in every communication event.



Basic MPI functions and commands

Simple MPI program identifying processes

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank + 1, size);

    MPI_Finalize();
    return 0;
}
```



Basic requirements for an MPI program

Basic requirements for an MPI program

- Include hearder file `mpi.h`.
- `MPI_Init(&argc, &argv)` starts MPI and a process group --- MPI_COMM_WORLD
- `MPI_Finalize()` exits MPI
- The group of processes in the program is called `MPI_COMM_WORLD`
  - `MPI_Comm_size(MPI_COMM_WORLD, &size)` gets the size of the group, i.e., # of processes.
- Each process is identified by its *rank*.
  - `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` gets the rank of the caller process.

Compile and Run a MPI program

- Compilation:
  - Regular applications: `gcc test.c -o test`
  - MPI applications: `mpicc test.c -o test`
- Execution:
  - Regular applications: `./test`
  - MPI applications (running with N processes):

```
$mpirun –n 4 ./test
I am 1 of 4
I am 2 of 4
I am 3 of 4
I am 4 of 4
```

Run a MPI program on multiple computers

- The program must be accessible by all computers with the same pathname.
  - Use a NFS or manually copy the file (scp).
- The user can remotely run a program on these computers using ssh
  - Better to set up password-less ssh login on all computers.
- Run 16 processes on 4 computers (4 processes on each of h1 ~ h3)

```
$mpirun –hosts h1:4,h2:4,h3:4,h4:4 –n 16 ./test
```

  - Runs the first four processes on h1, the next four on h2, etc.

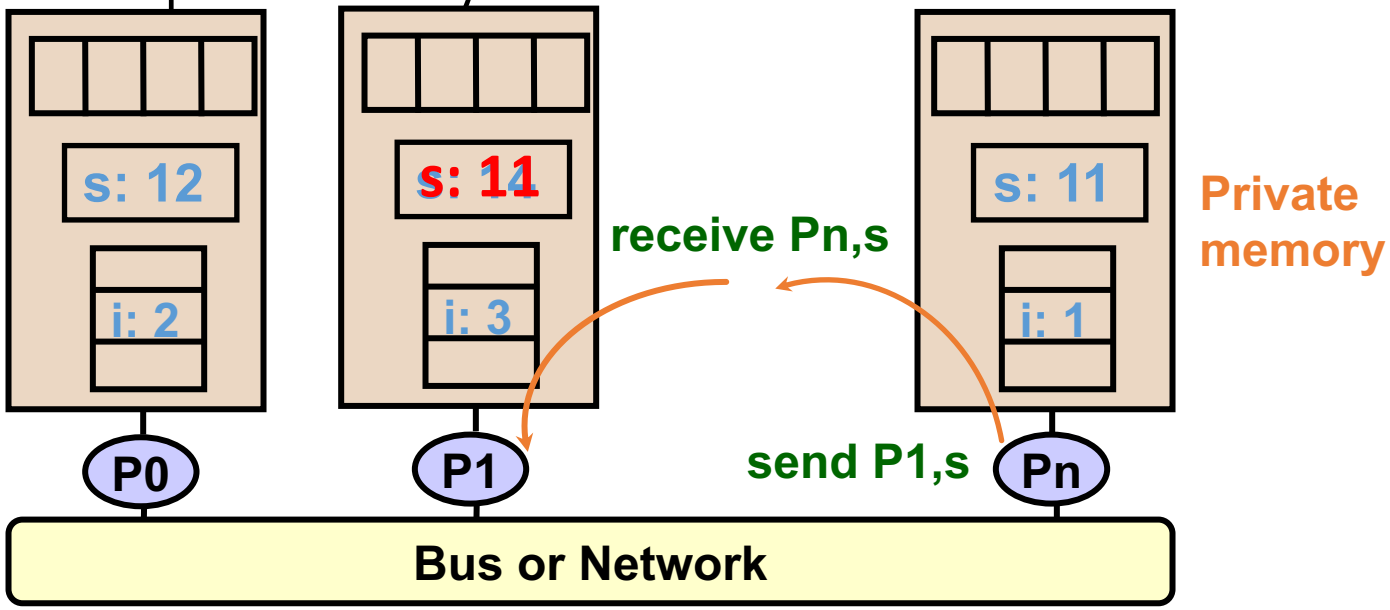
```
$mpirun –hosts h1,h2,h3,h4 –n 16 ./test
```

  - Runs the first process on h1, the second on h2, etc., and wraps around
  - So, h1 will have the 1st, 5th, 9th and 13th processes
- If there are many nodes, it might be easier to create a host file

```
$cat hf
   h1:14
   h2:2
$mpirun –hostfile hf ./test
```

MPI program

- A program consists of a collection of processes.
  - Usually created at program startup time and kept fixed during execution
- SPMD (Single Program Multi-Data)
  - Multiple processes execute the same copy of code
  - Each process is a thread of control with its own local address space.
- NO shared data between processes.
  - Processes communicate by explicit `send` and `receive` function pairs
  - Coordination is implicit in every communication event.



Single Program Multi-Data (SPMD)

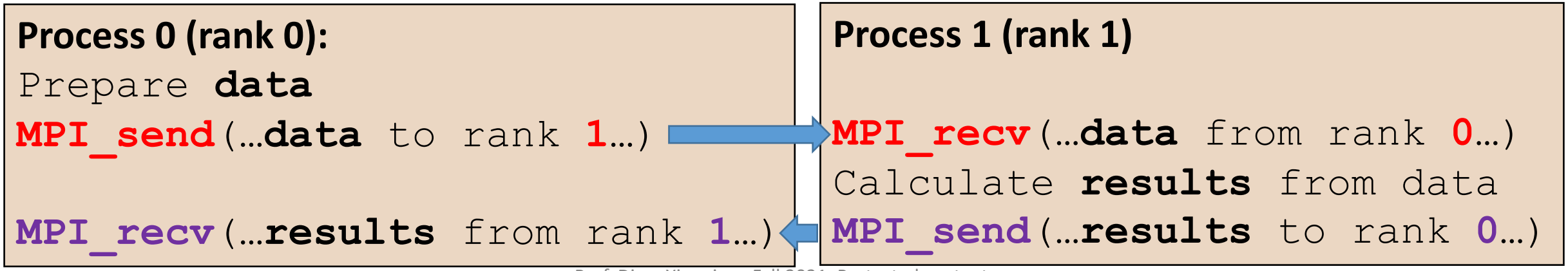
- Multiple processes share the same copy of code.
  - Program is actually launched separately on different computers.
- Each process is a thread of control with its own local address space.
  - Each process has a copy of `rank`.
  - `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` saves different values to different `rank` copies in different processes.
  - `printf()` prints different `rank` values by different processes.
  - Though different processes have different copies of `size` variable, `MPI_Comm_size (MPI_COMM_WORLD, &size)` saves the same value into them.
- Different processes can execute different code based on their ranks.
- Different processes can exchange data using messages.
  - `MPI_Send()`
  - `MPI_Recv()`
  - `...`

Different processes can execute different code and exchange data

```
if (rank == 0) { // code executed by process 0
  prepare data
  MPI_Send(..data to rank 1..)
  MPI_Recv(..results from rank 1..)
else if (rank == 1) { //code for process 1
  MPI_Recv(..data from rank 0..)
  compute results
  MPI_Send(..results to rank 0..)
}
```

- MPI_Send() sends a message
- MPI_Recv() waits and receives a message
- MPI_Send() and MPI_Recv() must be properly paired for smooth execution

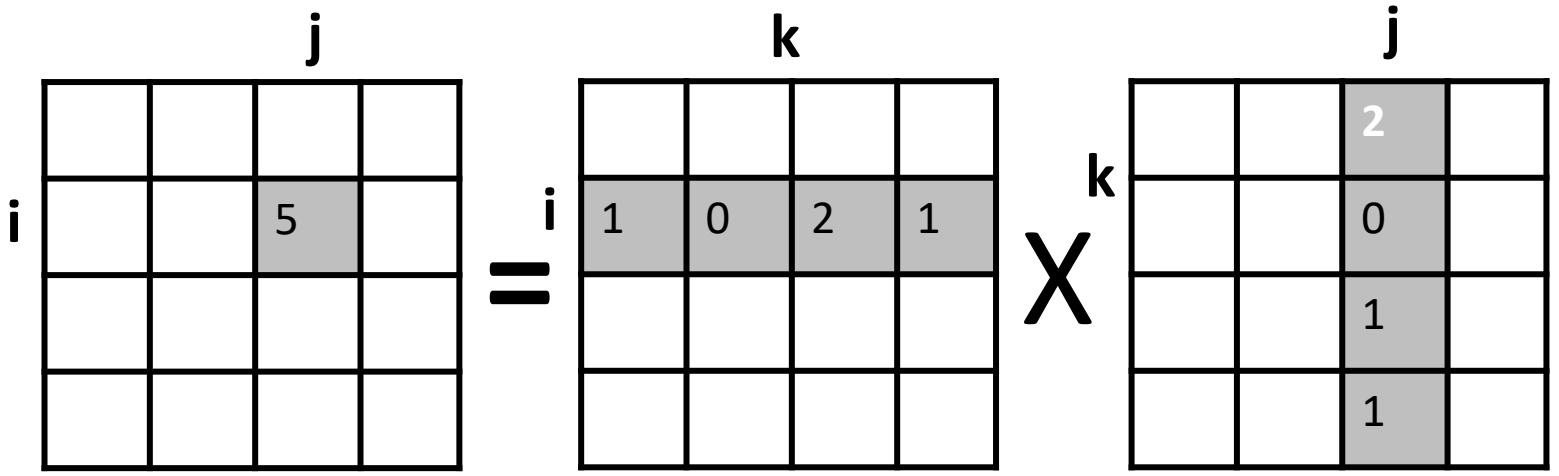
Offload some computation from process 0 to process 1



Matrix multiplication in parallel

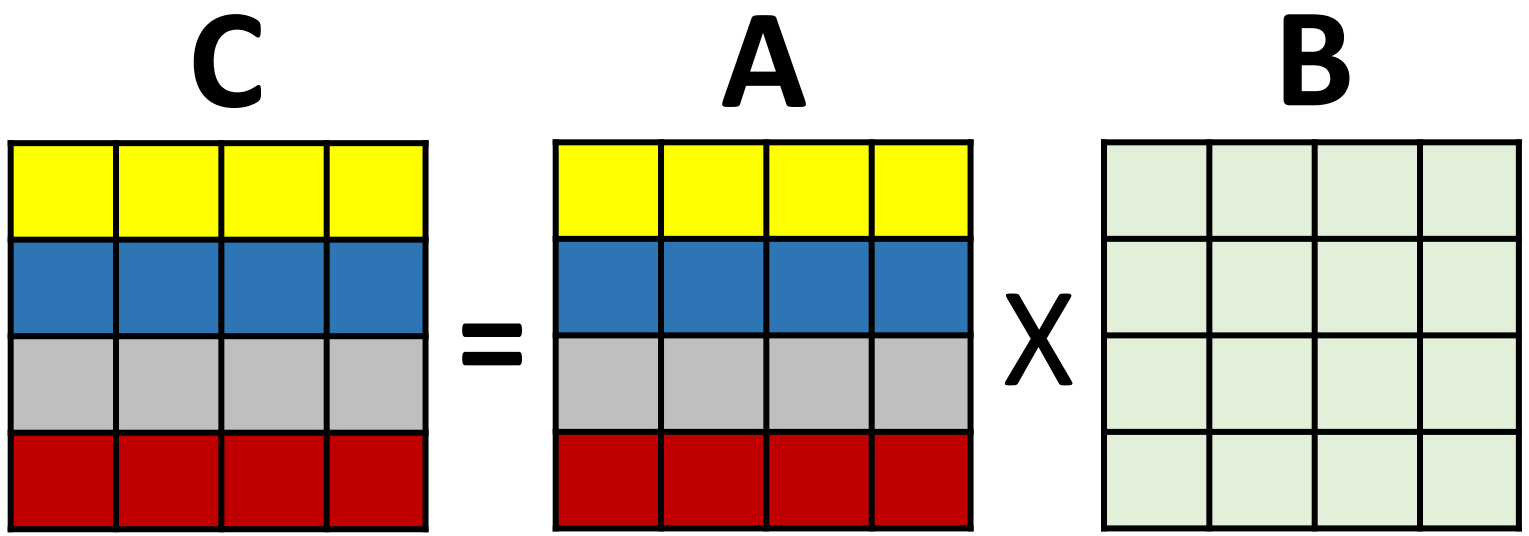
all computation

```
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      C[i][j]+=A[i][k]
              *B[k][j]
```



computation of each worker

```
for (i=low;i<high;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      C[i][j]+=A[i][k]
              *B[k][j]
```



Matrix multiplication in parallel (1 master multiple workers)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 100                  /* size of each dimension */
#define FROM_MASTER 1          /* setting a message type */
#define FROM_WORKER 2          /* setting a message type */

int main (int argc, char *argv[]){
   int     numtasks, taskid, numworkers, rows, i, j, k;
   double a[N][N], b[N][N], c[N][N]; /*matrices */
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   numworkers = numtasks-1;
   rows = N/numworkers;
```

```
if (taskid == 0)    { /* code for master */
    /*Initializing arrays*/
    for (i=0; i<N; i++)
      for (j=0; j<N; j++) {  a[i][j]= i+j;  b[i][j]= i*j; }

    /* Send matrix data to the worker tasks */
    for (int dest=1; dest<=numworkers; dest++)      {
       MPI_Send(&a[(dest-1)*rows][0], rows*N,
                MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD);
       MPI_Send(&b, N*N, MPI_DOUBLE, dest,
                   FROM_MASTER, MPI_COMM_WORLD);
    }
    /* Receive results from worker tasks */
    for (int source=1; source<=numworkers; source++)
       MPI_Recv(&c[(source-1)*rows][0],rows*N, MPI_DOUBLE,
                  source, FROM_WORKER, MPI_COMM_WORLD, &status);

    printf("*********  Result Matrix **********\n");
```

```
      for (i=0; i<N; i++) {
         for (j=0; j<N; j++) printf("%6.2f   ", c[i][j]);
         printf("\n");
      }
   }
   if (taskid > 0) { /* code for worker */
      MPI_Recv(&a, rows*N, MPI_DOUBLE, 0, FROM_MASTER,
               MPI_COMM_WORLD, &status);
      MPI_Recv(&b, N*N, MPI_DOUBLE, 0, FROM_MASTER,
               MPI_COMM_WORLD, &status);
      for (i=0; i<rows; i++)
        for (j=0; j<N; j++)
          for (k=0; k<N; k++)
            c[i][k] = c[i][k] + a[i][j] * b[j][k];

      MPI_Send(&c, rows*N, MPI_DOUBLE, 0, FROM_WORKER,
               MPI_COMM_WORLD);
   }
   MPI_Finalize();
}
```

MPI_Send(void ***buf**, int **count**, MPI_Datatype **datatype**, int **dest**, int **tag**,
MPI_Comm **comm**)

MPI_Recv(void *buf, int **count**, MPI_Datatype **datatype**, int **src**, int **tag**,
MPI_Comm **comm**, MPI_Status ***stat**)

- **buf**, **count**, and **datatype** determine what data would be sent/received.
- Information for pairing sender and receiver: **comm**, **dest**, **src**
  - Use `MPI_COMM_WORLD` as **comm**.
  - For sender process, **dest** in MPI_Send() is where should the data be sent to (i.e., receiver process).
  - For receiver process, **src** in MPI_Recv() is where should it expect the data coming from (i.e., sender process).
- **tag** is a user-defined message identification number (integer) for pairing MPI_Send() and MPI_Recv() calls.
- When MPI_Send() returns, the data has been delivered to the system and the buffer can be reused.
- When MPI_Recv() returns, the data is received and saved in buf.

Use MPI data types for communications

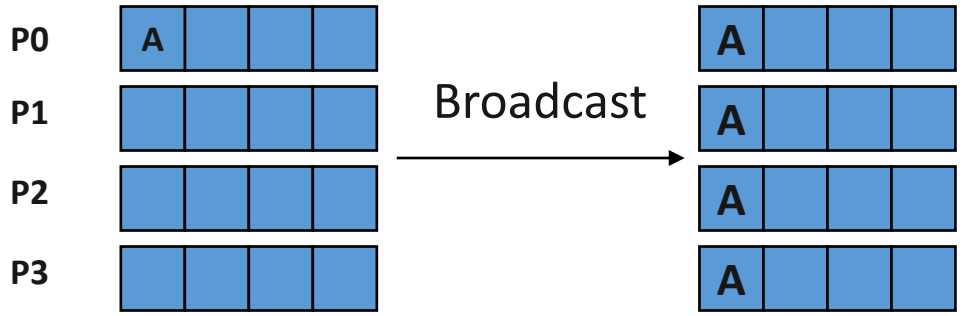
MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI datatype is very similar to a C

MPI_Send/MPI_Recv can be replaced by collective operations

MPI_Bcast(void ***buffer**, int **count**, MPI_Datatype **type**, int **root**, MPI_Comm **comm**)

- broadcasts message from root to all processes (including root).
- Called by all processes
- INOUT : buffer (starting address)
- IN : count (num entries in buffer)
- IN : type (type of each entry)
- IN : root (rank of broadcast root)
- IN : comm (communicator, MPI_COMM_WORLD)
- Improve both simplicity and efficiency.
- On return, contents of buffer is copied to all processes.



```
MPI_Reduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm)
```

- IN sendbuf (address of send buffer)
- OUT recvbuf (address of receive buffer)
- IN count (number of elements in send buffer)
- IN datatype (data type of elements in send buffer)
- IN op (reduce operation)
- IN root (rank of root process)
- IN comm (communicator)

- MPI_Reduce combines elements specified by send buffer and performs a **reduction operation** op on them.

MPI built-in collective computation operations

- **MPI_MAX** Maximum
- **MPI_MIN** Minimum
- **MPI_PROD** Product
- **MPI_SUM** Sum
- **MPI_LAND** Logical and
- **MPI_LOR** Logical or
- **MPI_LXOR** Logical exclusive or
- **MPI_BAND** Bitwise and
- **MPI_BOR** Bitwise or
- **MPI_BXOR** Bitwise exclusive or
- **MPI_MAXLOC** Maximum and location
- **MPI_MINLOC** Minimum and location

Example: Calculating the approximation of pi

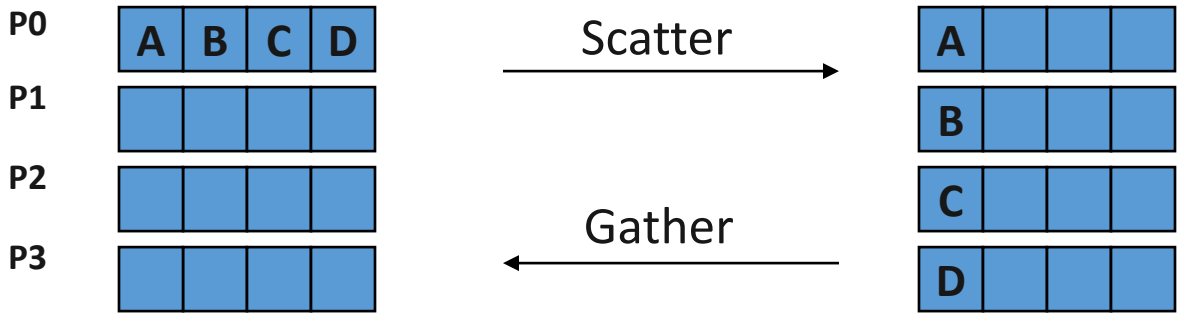
$$PI = \lim_{n\to\infty}\left(\frac{1}{n}\sum_{i=1}^{n}\frac{4.0}{1.0+\left(\frac{i-0.5}{n}\right)^2}\right)$$

```
#include "mpi.h"
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264643;
    double pi, sum, partial_sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
    if (myid == 0) {
        printf("Enter the number of terms:");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    partial_sum = 0.0;
    for(i=n/numprocs*myid+1; i<= n/numprocs*(myid+1); i++) {
        x = ((double)i - 0.5)/n;
        partial_sum+= 4.0 / (1.0 + x*x);
    }

    MPI_Reduce(&partial_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,
                    MPI_COMM_WORLD);

    if (myid == 0){
        pi = sum/n;
        printf("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
    return 0;
}
```

MPI_Gather and MPI_Scatter

- MPI_Scatter() distribute data from one process to all other processes in a communicator
  - Inverse of MPI_Gather
  - Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter.
- MPI_Gather() gathers together values from a group of processes
  - Each process sends content of send buffer to the root process.
  - Root receives and stores in rank order.



```
MPI_Scatter(void *sendbuf,int sendcount,MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

- IN sendbuf (starting address of send buffer)
- IN sendcount (number of elements **sent to each process**)
- IN sendtype (type)
- OUT recvbuf (address of receive bufer)
- IN recvcount (n-elements **in receive buffer**)
- IN recvtype (data type of receive elements)
- IN root (rank of sending process)
- IN comm (communicator)

Note: **send_buf, sendcount, sendtype** arguments ignored for non-root processes.

```
MPI_Gather(void *sendbuf,int sendcount,MPI_Datatype sendtype,
           void *recvbuf,int recvcount,MPI_Datatype recvtype,
           int root, MPI_Comm comm)
```

- IN sendbuf (starting address of send buffer)
- IN sendcount (number of elements **in send buffer**)
- IN sendtype (type)
- OUT recvbuf (address of receive bufer)
- IN recvcount (n-elements **for any single receive**)
- IN recvtype (data type of recv buffer elements)
- IN root (rank of receiving process)
- IN comm (communicator, MPI_COMM_WORLD)

Note: **Recvbuf** argument ignored for all non-root processes (also recvtype, etc.) **recvcount** on root indicates number of items received from each process, not total. This is a very common error.

Matrix multiplication in parallel (each process works on one data piece)

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define N 100                   /* size of each dimension */

int main (int argc, char *argv[]) {
  intnumtasks, taskid,rows, i, j, k;
  double a[N][N], b[N][N], c[N][N], aprime[N][N], cprime[N][N];
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

  if (taskid == 0)    {
      printf("run with %d tasks.\n",numtasks);
      for (i=0; i<N; i++) /*Initializing arrays*/
         for (j=0; j<N; j++) { a[i][j]= i+j; b[i][j]= i*j; }
  }
```

```
  rows = N/numtasks;
  MPI_Bcast(b, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(a, rows * N, MPI_DOUBLE, aprime, rows * N, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

  for (i=0; i<rows; i++)
    for (k=0; k<N; k++)
      for (j=0; j<N; j++)
        cprime[i][k] = cprime[i][k] + aprime[i][j] * b[j][k];


  MPI_Gather(cprime, rows * N, MPI_DOUBLE, c, rows * N, MPI_DOUBLE,
          0, MPI_COMM_WORLD);

  if(taskid == 0) {
    printf("************  Result ****************\n");
    for (i=0; i<N; i++) {
       for (j=0; j<N; j++) printf("%6.2f  ", c[i][j]);
       printf("\n");
    }
  }
  MPI_Finalize();
}
```

Avoid deadlock

- Process may wait in MPI_Send() and MPI_Recv().
- Deadlock may be caused if they are not placed in a correct order.

Process 0 (rank 0):

```
MPI_recv(…B from rank 1…)
MPI_send(…A to rank 1…)
```

Process 1 (rank 1)

```
MPI_recv(…A from rank 0…)
MPI_send(…B to rank 0…)
```

CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

Data in a file is also a bit string (no type info)

Check file content using hexadecimal format

Vim:%!xxd to switch to hex, :%!xxd -r to switch back

address area **hexadecimal area** **character area**

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 5058 0000 0000 0000  ..>.....PX......
00000020: 4000 0000 0000 0000 a003 0200 0000 0000  @...............
00000030: 0000 0000 4000 3800 0900 4000 1c00 1b00  ....@.8...@.....
00000040: 0600 0000 0500 0000 4000 0000 0000 0000  ........@.......
00000050: 4000 0000 0000 0000 4000 0000 0000 0000  @.......@.......
00000060: f801 0000 0000 0000 f801 0000 0000 0000  ................
```

Data contents in /bin/ls

vi /usr/share/doc/gcc/README.Debian

What does it look like normally:

```
            The Debian GNU Compiler Collection Setup
            ========================================

Abstract
--------

Debian uses a default version of GCC for most packages; however, some
```

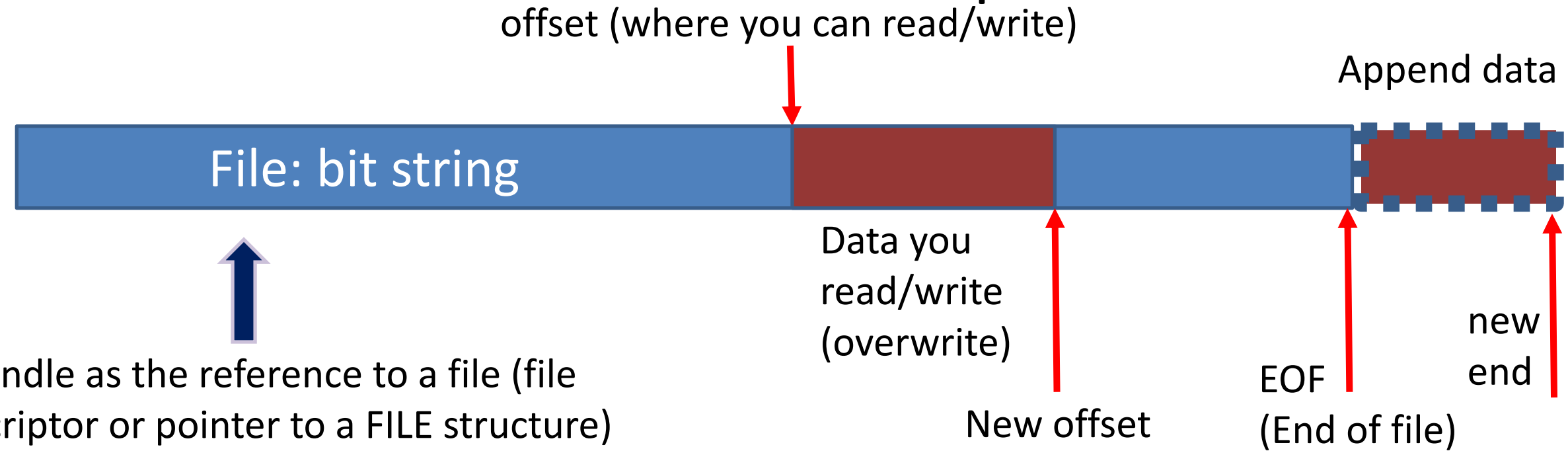
What is saved in the file:

```
00000000: 0909 5468 6520 4465 6269 616e 2047 4e55  ..The Debian GNU
00000010: 2043 6f6d 7069 6c65 7220 436f 6c6c 6563   Compiler Collec
00000020: 7469 6f6e 2053 6574 7570 0a09 093d 3d3d  tion Setup...===
00000030: 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d  ==============
```

Outline

- File operations
- handle directory entries
- Travels a directory

Overview of files and file operations



Step 1: Open a file

- Information needed: pathname and mode
- Can be used to create a file
  - Additional info needed: file permissions.
- Return a handle

Step 3: close a file

Step 2: Read/write a file

- Happens at the current file offset
  - Change file offset when necessary
- When offset is in the middle of the file
  - returns the number of bytes
  - Overwrites data
  - offset updated automatically
- When offset is at the end of a file, reads return EOF and writes append

Library functions for opening, creating, and closing a file

```
#include <stdio.h>
FILE * fopen (const char *path, const char *mode);
int fclose(FILE *stream);
```

```
FILE *stream;
stream = fopen ("/etc/manifest", "r");
if (stream == NULL)
    perror("fopen");
/* read data */
fclose(stream);
```

- return a pointer to FILE structure (FILE *)
  - return NULL if file is not opened.
- If used to create a file, file permission is 666.

Mode is a string

- r Open the file for reading.
  - The file offset is initialized to 0.
- r+ Open the file for both reading and writing.
  - The file offset is initialized to 0.
- w Open the file for writing.
  - If the file exists, it is truncated to zero length.
  - If the file does not exist, it is created.
  - The file offset is initialized to 0
- w+ Open the file for both reading and writing.
  - If the file exists, it is truncated to zero length.
  - If the file does not exist, it is created.
  - The file offset is initialized to 0
- a Open the file for appending.
  - If the file does not exist, it is created.
  - The file offset is initialized to the end of the file
- a+ Open the file for reading and appending.
  - If the file does not exist, it is created.
  - The file offset for reading is initialized to 0
  - Output is always written at the end of the file

Library functions for changing file offset

```
#include <stdio.h>
int fseek (FILE *stream, long pos, int origin);
/* to get the current offset */
long ftell (FILE *stream);
/* to reset current offset to beginning of file */
void rewind(FILE *stream);
```

- the `origin` argument can be one of the following:
  - `SEEK_CUR`: new file offset = current value + pos
    - When pos == zero, lseek returns the current file offset.
  - `SEEK_END`: new file offset = the current length of the file + pos.
  - `SEEK_SET`: new file offset = pos.
- `pos` can be positive, zero, or negative.

Library functions for reading/writing a file

```
#include <stdio.h>
/* read/write raw data */
size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
size_t fwrite (void *buf, size_t size, size_t nr, FILE *stream);
/* read/write chars/strings */
int getc(FILE *stream);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int fputc (char c, FILE *stream);
int fputs (const char *str, FILE *stream);
/* Read and convert into text */
int fscanf(FILE *stream, const char *format, ...);
/* convert into text and then write */
int fprintf(FILE *stream, const char *format, ...);
```

A few questions on binary file vs. text file

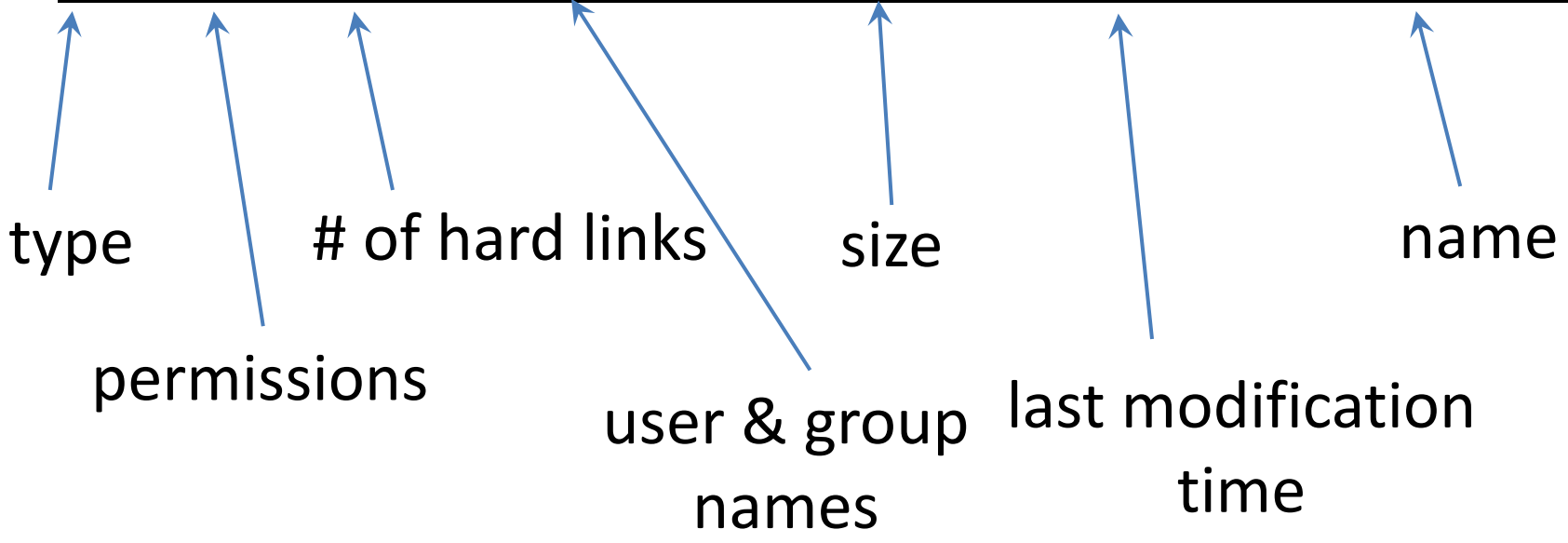
- How can you tell whether a file is a binary file or text file?
- When you create a file, which format do you choose, binary or text?
- When you read/write a binary file, which functions do you choose?
- When you read/write a text file, which functions do you choose?

Outline

- File operations
- Handle directory entries
- Travels a directory

File and directory information

```
ubuntu@ip-172-30-0-5:~/temp$ ls -l
total 232
drwxrwx--- 2 ubuntu ubuntu   4096 Jan 24 05:13 bak
-rw--w---- 1 ubuntu ubuntu 116181 Nov 20 20:06 stock.html
-rw--w---- 1 ubuntu ubuntu  90722 Jul  8  2011 tagsoup-1.2.1.jar
-rwxrwx--- 1 ubuntu ubuntu  13728 Dec 15 19:38 test
-rw-rw---- 1 ubuntu ubuntu   2324 Dec 15 19:23 test.c
```



Directory and directory entry

Directory entry

```
struct dirent {
  ino_t  d_ino;
  /* inode number */
  char  d_name[];
  /* filename */
  int len;
  …
};
```

Directory: a set of directory entries

```
struct dirent
entry1={123, ".", …}
```

```
struct dirent
entry2={234, "..", …}
```

```
struct dirent
entry3={235, "photos", …}
```

```
struct dirent
entry4={236, "sort.c", …}
```

Methods for accessing directory entries

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

```
struct dirent {
   ino_t  d_ino;
   /* inode number */
   char  d_name[256];
   /* filename */
   …
};
```

To get all the entries in a directory

- open a directory
- read out an entry (repeat until readdir returns NULL)
- Close the directory.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc, char **argv){
        struct dirent *pent;   DIR *pdir;

        if(argc!=2 || strlen(argv[1])==0) {
                printf("%s dir\n", argv[0]);  exit(1);
        }

        pdir = opendir(argv[1]);
        if(pdir==NULL) {
                printf("open directory fails.\n");  exit(1);
        }

        while( (pent=readdir(pdir))!=NULL )
                printf("%s\n", pent->d_name);

        closedir(pdir);
}
```

```
$ ./myls /bin
.
..
grep
loadkeys
btrfs-image
gendata
sync
```

Obtaining file information other than names

```
#include <sys/stat.h>
int stat( const char* pathname, struct stat* buf )
```

struct stat has the information about a file

NAME	MEANING
st_dev	device number
st_ino	file ID -- inode number
st_mode	type and permission flags
st_nlink	hard-link count
st_uid	user ID
st_gid	group ID
st_size	file size
st_atime	last access time
st_mtime	last modification time
st_ctime	last status-change time

Some predefined macros

- st_mode as argument and return true(a value of 1) for the following file types:

MACRO	RETURNS TRUE FOR FILE TYPE
S_IFDIR	directory
S_IFCHR	character special device
S_IFBLK	block special device
S_IFREG	regular file
S_IFFIFO	pipe

- The time fields decoded using the standard C library asctime() and localtime() functions.

Outline

- File operations
- Handle directory entries
- Travels a directory

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <time.h>

void traverse(char *pathName ) {/* traverse a directory */
     struct stat statBuf;
     mode_t mode;
     int result, charsRead;
     DIR *pDir;
     struct dirent *pEnt;
     char fileName[1024];

     result = stat(pathName, &statBuf); /* Obtain file status */
     if ( result == -1 ) { /* Status was not available */
          fprintf( stderr, "Cannot stat %s \n", pathName );
          return;
     }
```

```
    mode = statBuf.st_mode;  /* Mode of file */

    if(S_ISREG(mode)) /* Regular file */
         printf("%s: size %lu bytes, mod. time = %s",
              pathName, statBuf.st_size,
              asctime(localtime(&statBuf.st_mtime)));

    else if ( S_ISDIR( mode ) ) { /* Directory */
         printf("Entering directory %s\n", pathName);
         pDir=opendir(pathName); /* Open for reading */
         while( (pEnt = readdir(pDir)) != NULL ) {
              if ( strcmp(pEnt->d_name, ".") != 0 &&
                   strcmp( pEnt->d_name, "..") != 0) {
                   sprintf( fileName, "%s/%s", pathName,
                        pEnt->d_name );
                   traverse(fileName);
              }
         }
         closedir(pDir);
    }
}
```

```
int main(int argc, char **argv) {
     if(argc!=2 || strlen(argv[1])==0) {
          printf("%s dir\n", argv[0]);  exit(1);
     }

     traverse(argv[1]);
}
```

How to implement the following?

- Non-recursive DFS traversal
- Non-recursive BFS traversal
- Recursive BFS traversal

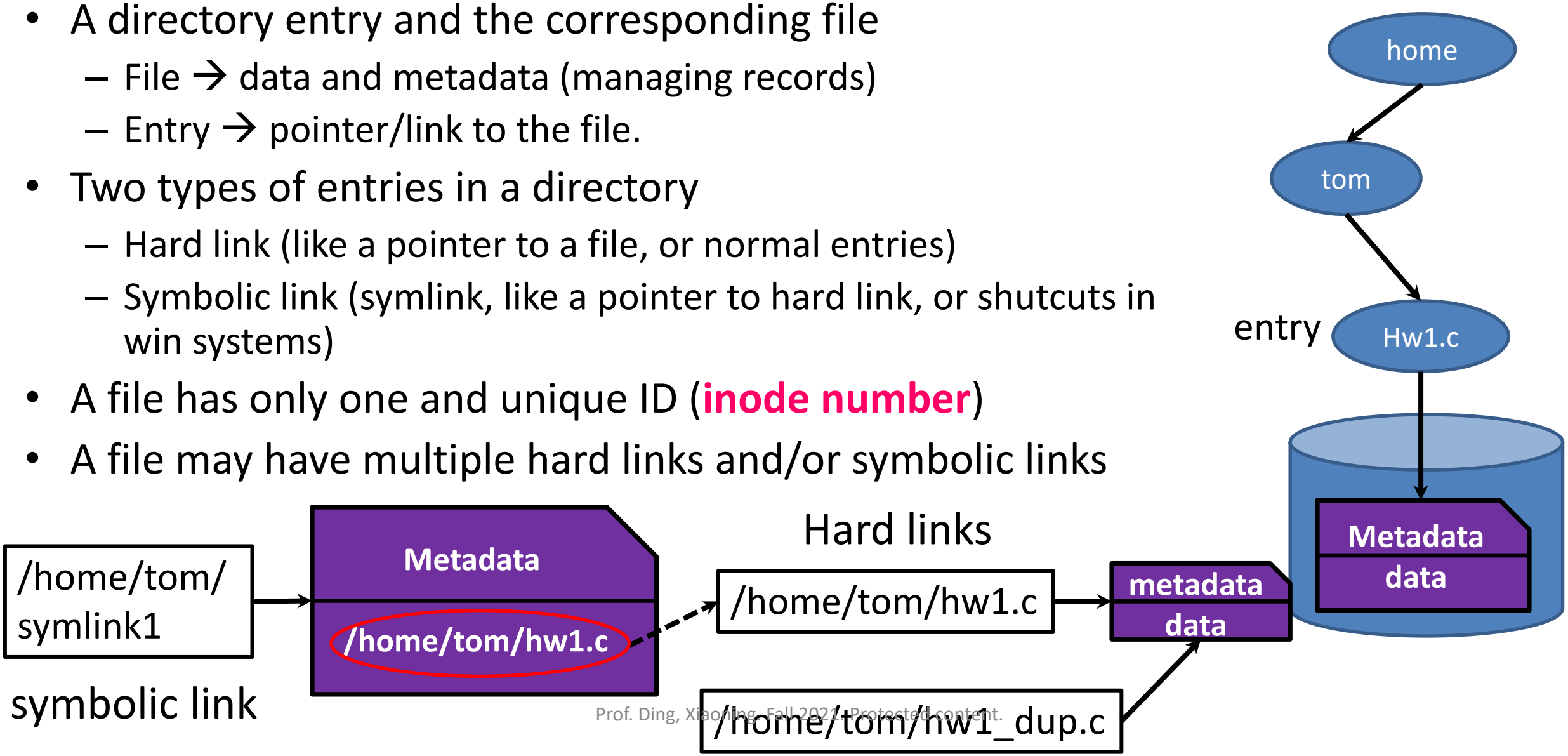
ADDITIONAL MATERIALS ON FILE/DIRECTORY OPERATIONS (NOT REQUIRED, ONLY FOR INTERESTED STUDENTS)

Other methods

- directory entries
  - link
  - unlink
  - symlink
  - rename
- file metadata
  - chown
  - chmod

What is a directory entry? Is it a file/subdirectory?

- A directory entry and the corresponding file
  - File → data and metadata (managing records)
  - Entry → pointer/link to the file.
- Two types of entries in a directory
  - Hard link (like a pointer to a file, or normal entries)
  - Symbolic link (symlink, like a pointer to hard link, or shutcuts in win systems)
- A file has only one and unique ID (**inode number**)
- A file may have multiple hard links and/or symbolic links



Creating a new hard link

```
int link( const char* oldPath, const char* newPath)
```

- link() creates a new entry *newPath* and links it to the same file to which the label *oldPath* is linked.
  - The first hard link of a file is created when the file is created (i.e., pathname in open, create, or fopen call)
- The hard-link count of the associated file is incremented by one.
- If *oldPath* and *newPath* reside on different physical devices, a hard link cannot be made and link() fails.
- returns -1 if unsuccessful and a value of 0 otherwise

```
% cat  mylink.c      ---> list the program.
main() {
     link("orginal.txt", "another.txt");
}

% cat original.txt  ---> list original file.
  this is a file.
% ls  -l   original.txt   another.txt   ---> another.txt not found
  -rw-r--r--  1  glass       16  May 25 12:18  original.txt
% mylink
% ls  -l  original.txt   another.txt   ---> examine files after.
  -rw-r--r--  2  glass       16  May 25 12:18  another.txt
  -rw-r--r--  2  glass       16  May 25 12:18  original.txt
% cat >> another.txt     ---> alter "another.txt".
  hi
  ^D
% ls  -l  original.txt   another.txt   ---> both labels reflect the change.
  -rw-r--r--   2   glass          20  May  25   12:19  another.txt
  -rw-r--r--   2   glass          20  May  25   12:19  original.txt
```

```
% rm original.txt       ---> remove original label.

% ls -l original.txt another.txt  ---> Only another.txt exists
-rw-r--r--   1   glass           20  May 25    12:19  another.txt

% cat  another.txt      ---> list contents via other label.
  this is a file.
  hi

%_
```

Symbolic links

- A symbolic link is an indirect pointer to a file – a pointer to the hard link to the file
  - A symlink is a special file containing a pathname
  - Like a shortcut in windows systems
- You can create a symbolic link to a directory
- A symbolic link can point to a file on a different file system
- A symbolic link can point to a non-existent file (aka. broken link)

```
#include <unistd.h>
int symlink(const char *target, const char
*linkpath);
```

Deleting a file: unlink()

```
int unlink( const char* fileName )
```

- unlink() removes a directory entry pointing to a file.
  - If *fileName* is the last hard link to the file, the file's resources are deallocated.
  - If *fileName* is a symbolic link to the file, the link is removed.
- If successful, unlink() returns a value of 0;
- otherwise, it returns a value of -1.

Changing a file's owner and/or group

```
int chown(const char* fileName, uid_t ownerId, gid_t  groupId)
int lchown(const char* fileName, uid_t ownerId, git_t  groupId)
int fchown(int fd,uid_t ownerId,gid_t groupId )
```

- chown() causes the owner and group IDs of fileName to be changed to ownerId and groupId, respectively.
  - A value of -1 in a particular field means that its associated value should remain unchanged.
- lchown():changes the ownership of a symbolic link
- fchown():takes an open descriptor as an argument instead of a filename.

Example, changed the group of the file “test.txt” from “music” to “cs” (group ID = 62).

```
%cat  mychown.c      ---> list the program.
main() {
   int flag;
   flag = chown(“test.txt”, -1, 62 ); /* user is ID uchanged */
   if ( flag == -1 ) perror(“mychown.c”);
}

% ls  -lg  test.txt      ---> examine file before the change.
 -rw-r--r--   1   glass     music     3  May 25 11:42   test.txt
% mychown              ---> run program.
% ls  -lg   test.txt     ---> examine file after the change.
 -rw-r--r--   1   glass     cs        3  May 25  11:42   test.txt
```

Changing file permissions

```
int chmod( const char* fileName, int mode )
int fchmod( int fd, mode_t mode );
```

- chmod() changes the mode of *fileName* to *mode*
- mode is usually supplied as an octal number,
- to change a file's mode, you must either own it or be a super-user.
- fchmod() takes an open file descriptor as an argument instead of a filename.
- both return a value of -1 if unsuccessful, and a value of 0 otherwise.

Example: changing the permission flags of the file “test.txt” to 600 octal (read and write permission for the owner only):

```
% cat mychmod.c    ---> list the program.
main() {
   int flag;
   flag = chmod(“test.txt”, 0600);  /* Use an octal encoding */
   if ( flag==-1 ) perror(“mychmod.c”);
}

% ls -l test.txt     ---> examine file before the change.
-rw-r--r--    1   glass          3   May  25  11:42   test.txt

% mychmod        ---> run the program.

% ls -l test.txt     ---> examine file after the change.
-rw-------    1   glass          3   May  25  11:42   test.txt
```

CS 288 Intensive Programming in Linux

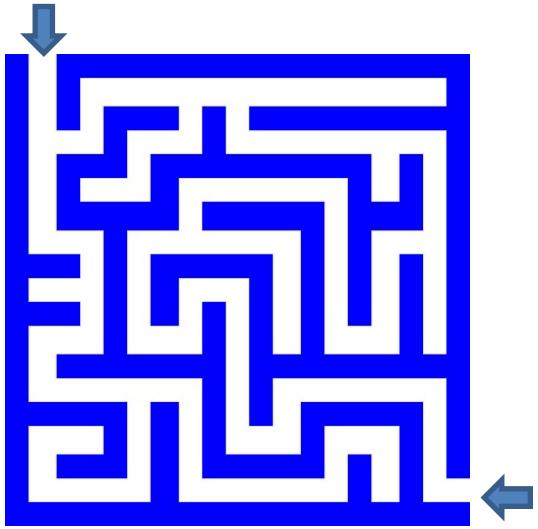
Professor Ding, Xiaoning

STATE SPACE SEARCH AND A* SEARCH

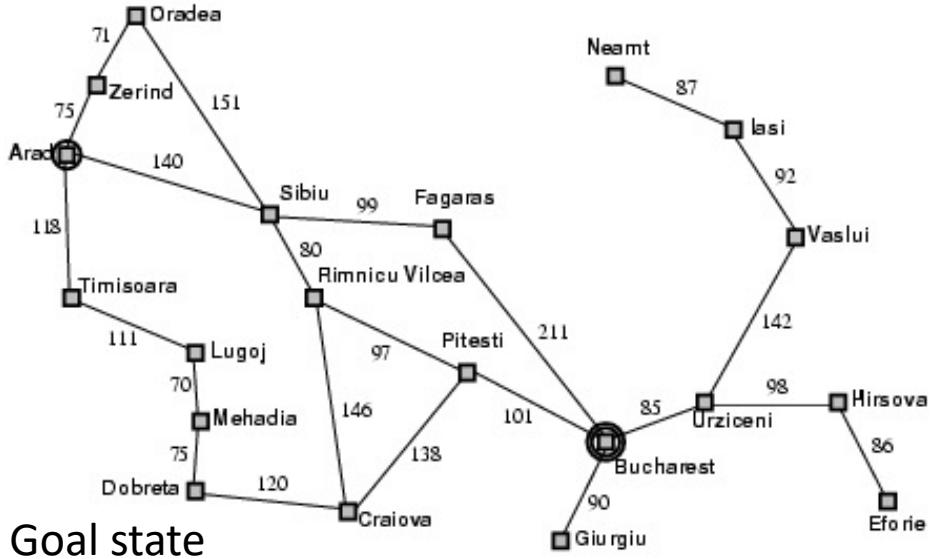
Solving problems by searching

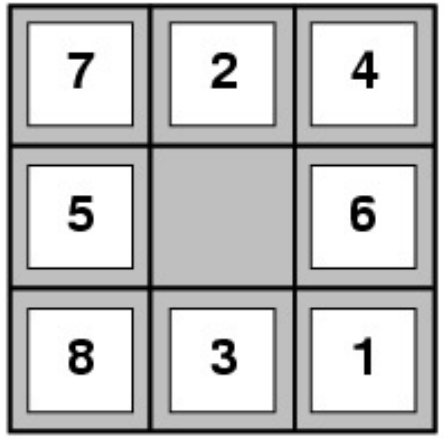
- The solution is a fixed sequence of actions
- Search is the process of looking for the sequence of actions that reaches the goal

Start state

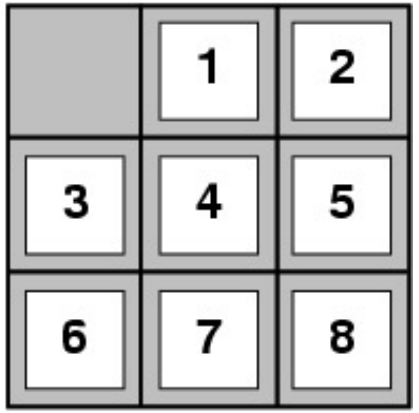


Goal state





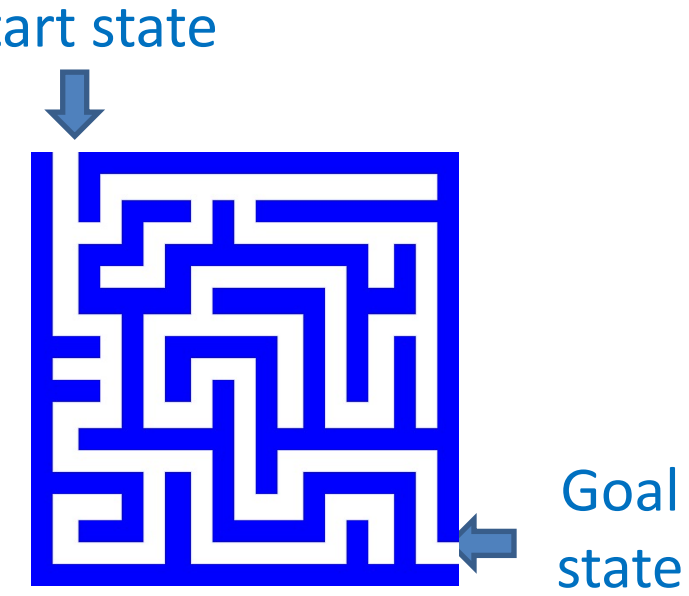
Start State



Goal State

factors in a search problem

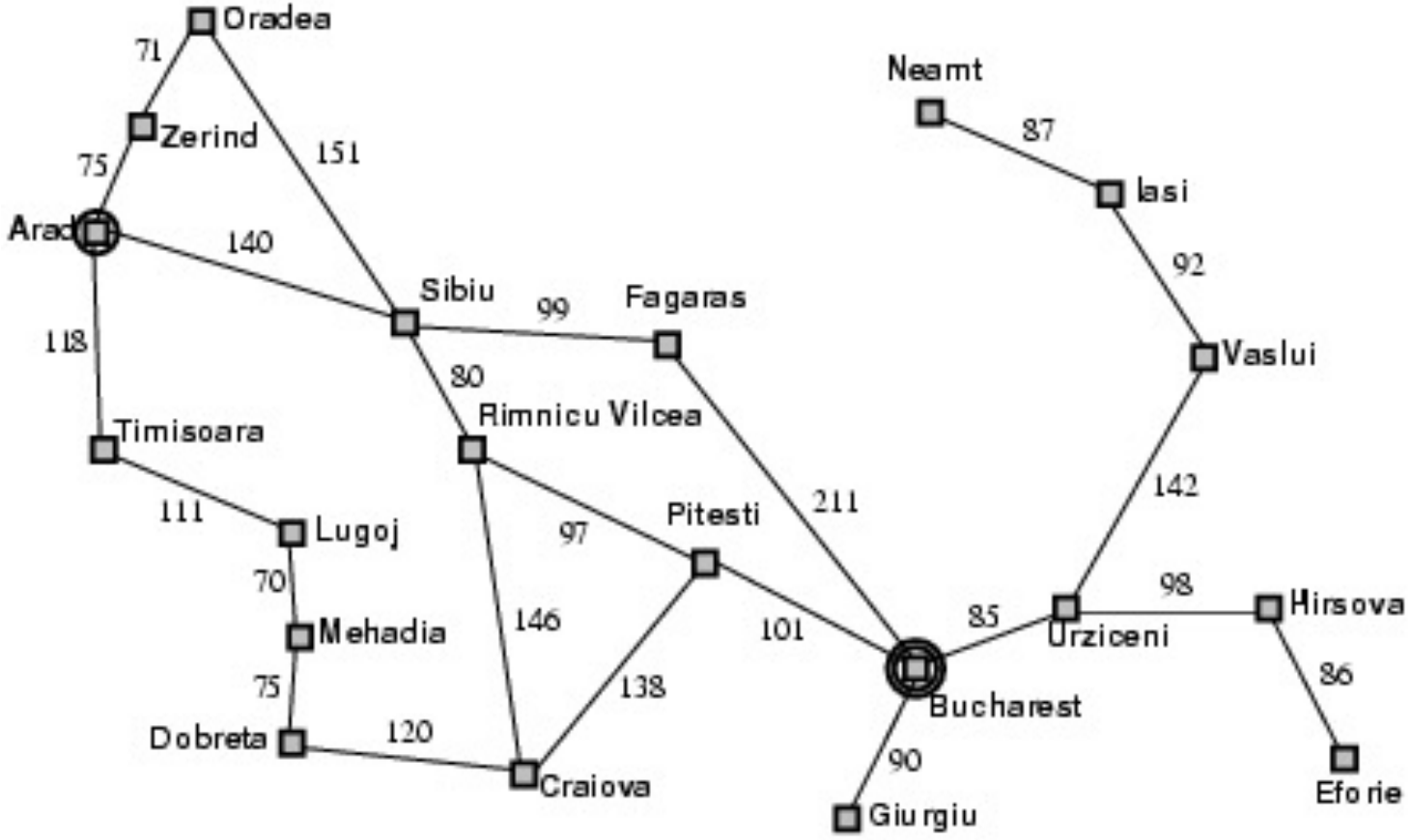
- **start state**
- **Actions**
- **Transition model**
  - What is the result of performing a given action in a given state?
- **Goal state**



- **A solution:** a sequence of actions that can change start state into goal state
- **Path cost:** total *cost* of a sequence of actions
- The **optimal solution** is the sequence of actions that have the lowest path cost (a problem may have multiple solutions)
- **Search algorithms**: how do we find high quality solutions?
  - How to find a solution if there are solutions?
  - How to find a low-cost solution?
  - How to find a low-cost solution with low cost?

Example: path finding

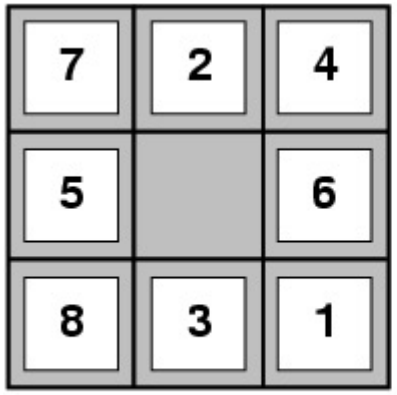
Find a path from *Arad* to *Bucharest*



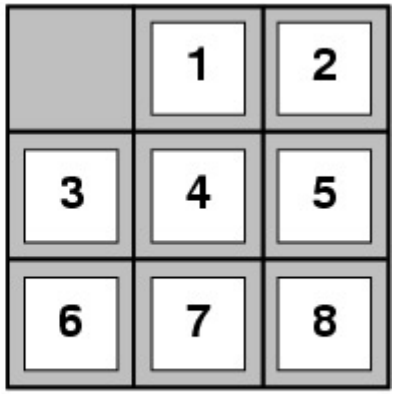
- **Start state :** Arad
- **Actions:** Go from one city to another
- **Transition model:** If you go from city A to city B, you end up in city B
- **Goal state:** Bucharest
- **Path cost :** Sum of distances

Example: the 8-puzzle

- **States:** Locations of tiles
  - 8-puzzle: 181,440 states
  - 15-puzzle: 1.3 trillion states
  - 24-puzzle: 10^{25} states
  - Optimal solution of n-Puzzle is NP-hard
- **Transition model**
  - The movement of the empty tile.
- **Actions**
  - Move empty tile left, right, up, down
- **Path cost**
  - 1 per move → # of moves



Start State



Goal State

Other real-world examples

- Routing
- Touring
- VLSI layout
- Assembly sequencing
- Protein design

Basic idea on searching for a solution

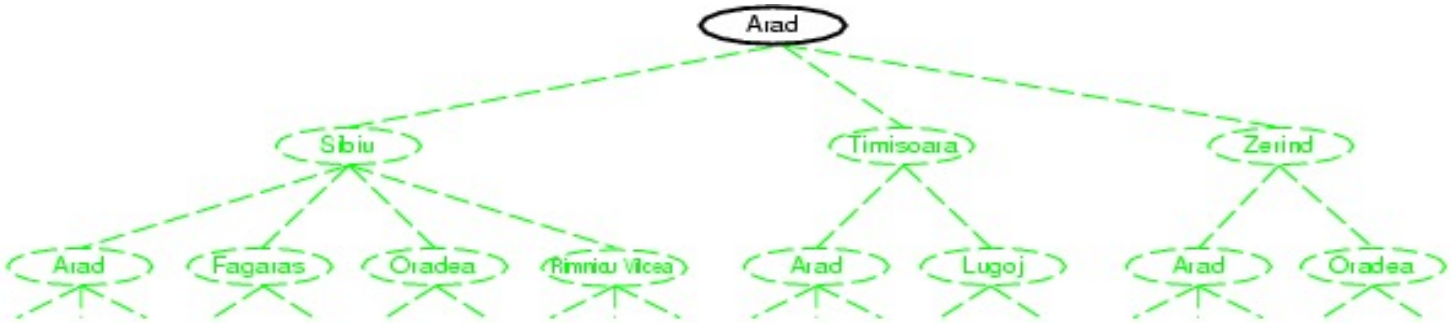
- Start from the **start state**
- Try different possible **actions**, and obtain **new states**
- Try different possible actions on new states, and obtain even more new states
- **Keep trying** until we find the goal state
- Check what actions were made to change the start state into the goal state → **solution**

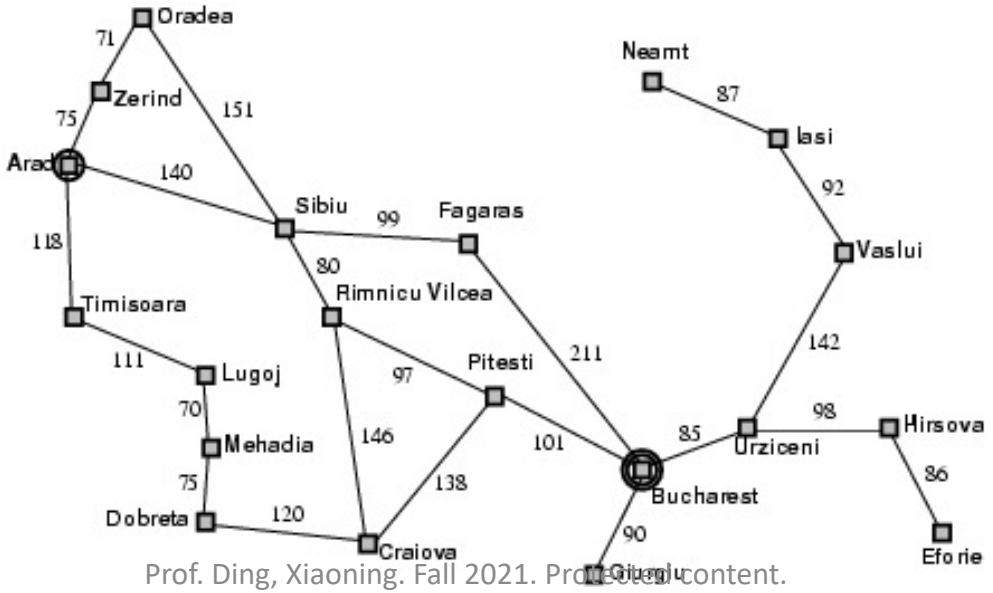
- **Searching** is actually the process of trying different actions and checking new states.

Search algorithm outline

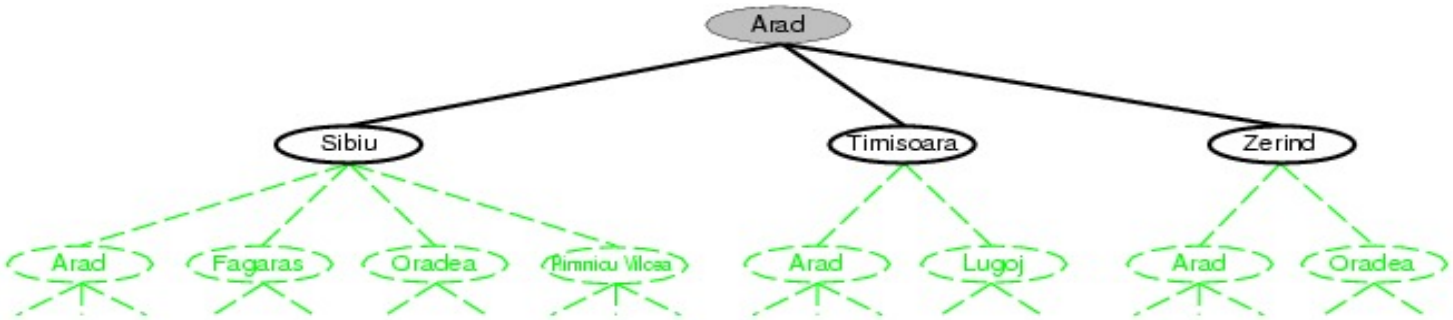
- The set of new states is called the **fringe.**
  - key data structure --- fringe : usually implemented as a **list**, and each state is a **node** on the list.
- the fringe = {**start state**}
- While the fringe is not empty
  - **Choose** a state and remove it from fringe
  - **Check** the state: if the node contains the **goal state**, return solution.
  - **Expand** the state: for each possible action, generate a successor (new?) state, add successor states into the fringe.

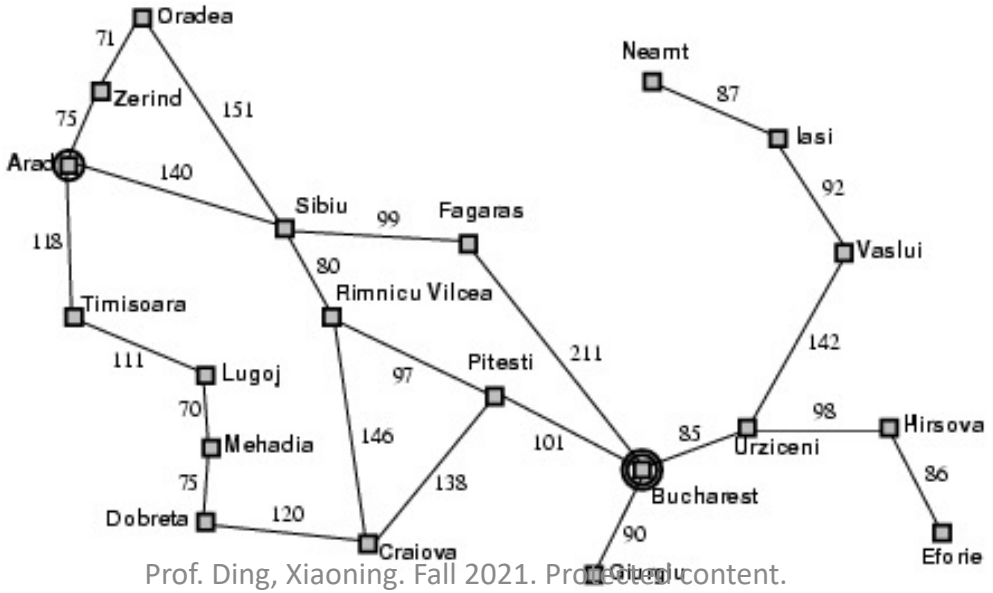
A search example



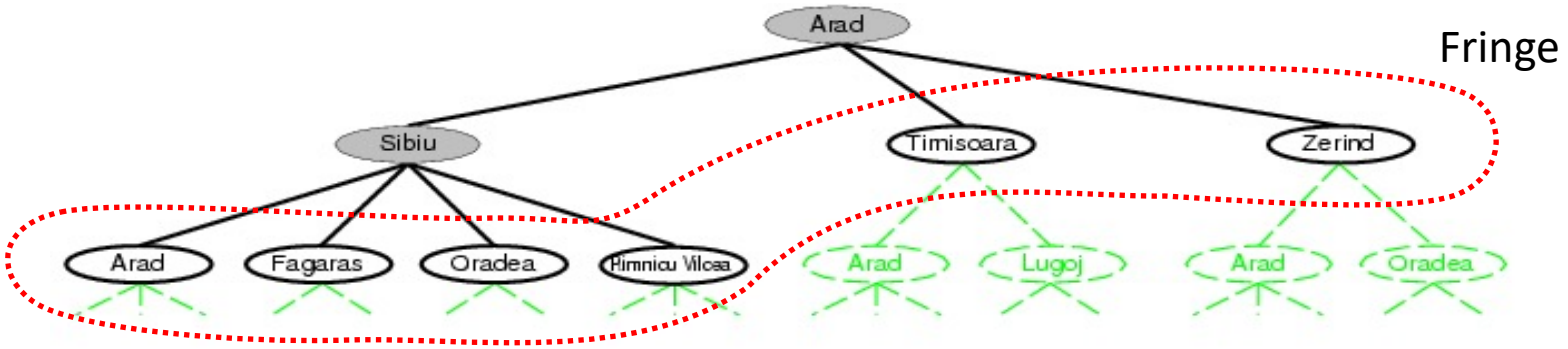


A search example

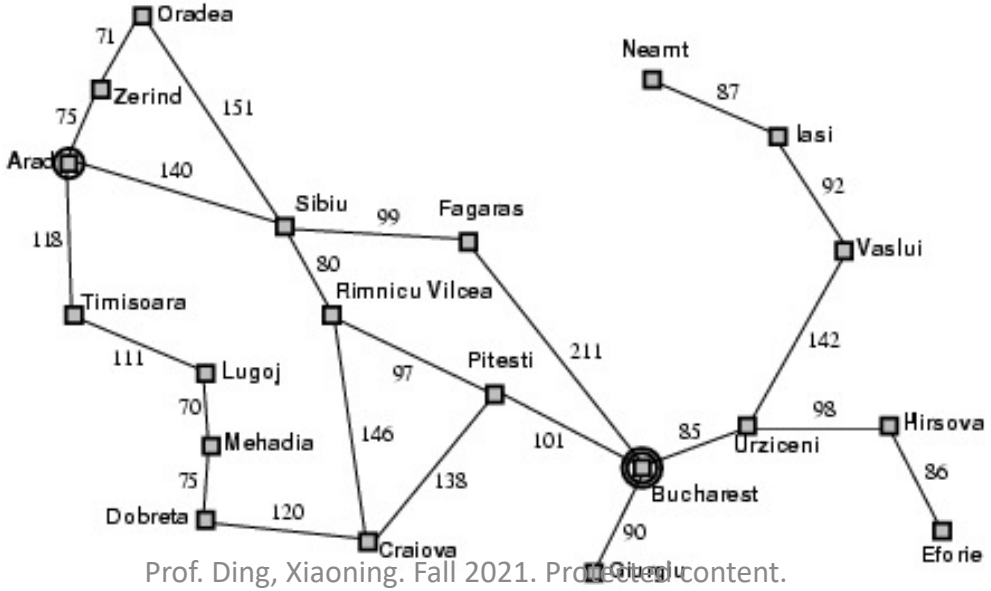




A search example

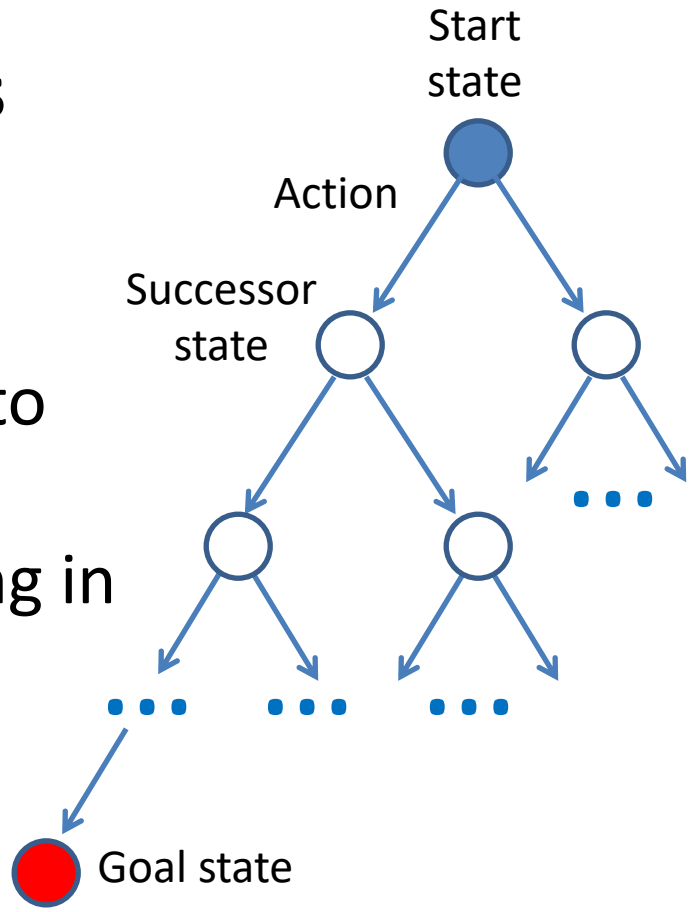


Fringe



Conceptually searching a solution is to traverse a tree of states

- Root node corresponds to the start state
- Children of a node correspond to it **successors states**
- Edges corresponds to actions
  - Depth of a node is how many actions were made to generate the corresponding state
  - A solution is the path starting from root and ending in a goal state
  - There may be multiple goal states. But the search can end when any one is reached.



Note: Implementation doesn't need to actually build the tree structure.

Search strategy

Search strategy --- which state in the fringe should be chosen to check/expand first?

- Determine whether a solution can be found.
- Determines quality of the solution.
- Determines overhead (execution time and memory space used to find a solution).

Uninformed and informed search strategies

Uninformed strategies consider all states are equally desirable.

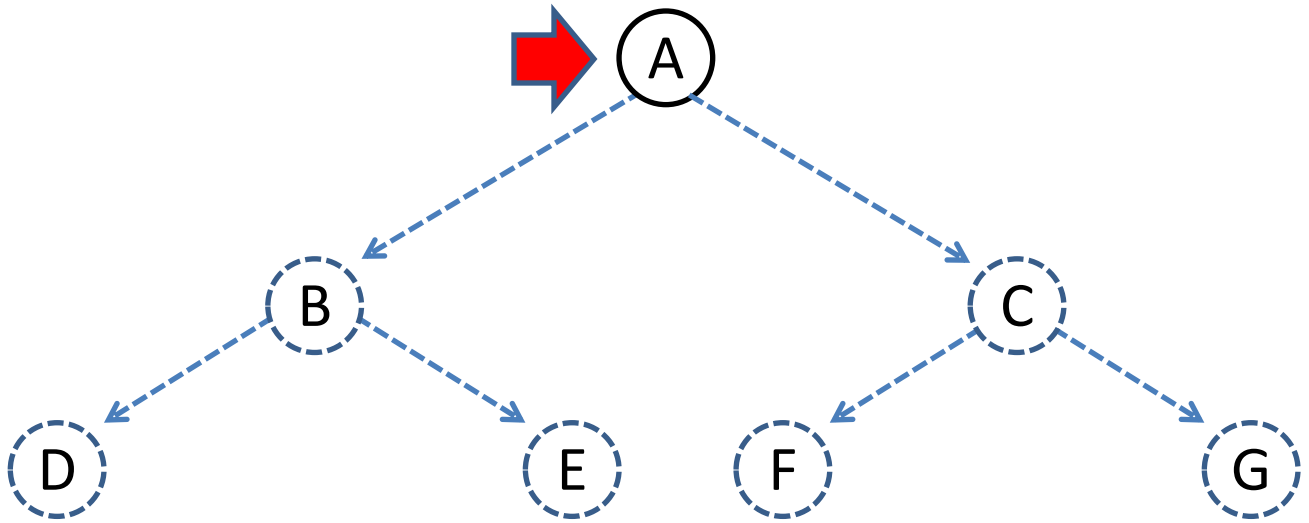
- use only the information available in the problem definition.
- Breadth-first search
- Depth-first search
- Iterative deepening search

Informed strategies give the algorithm "hints" about the desirability of different states

- Use an *evaluation function* to rank states and select the most promising one for expansion
- Greedy best-first search
- A* search

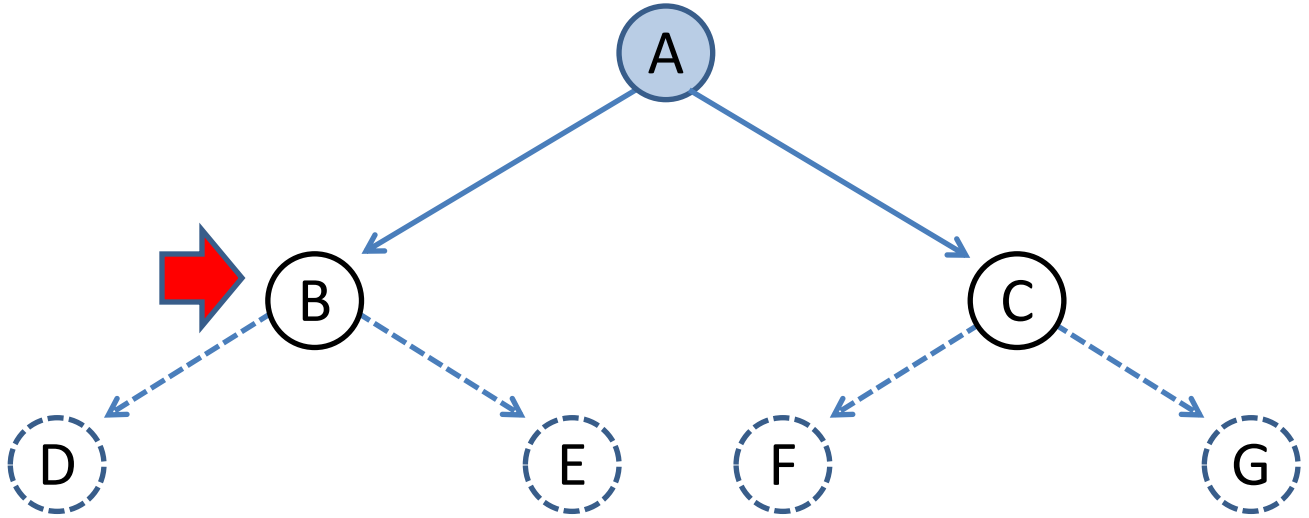
Breadth-first search strategy

- Expand the shallowest unexpanded state
  - Depth : how many actions were made to generate the corresponding state
- Non-recursive implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



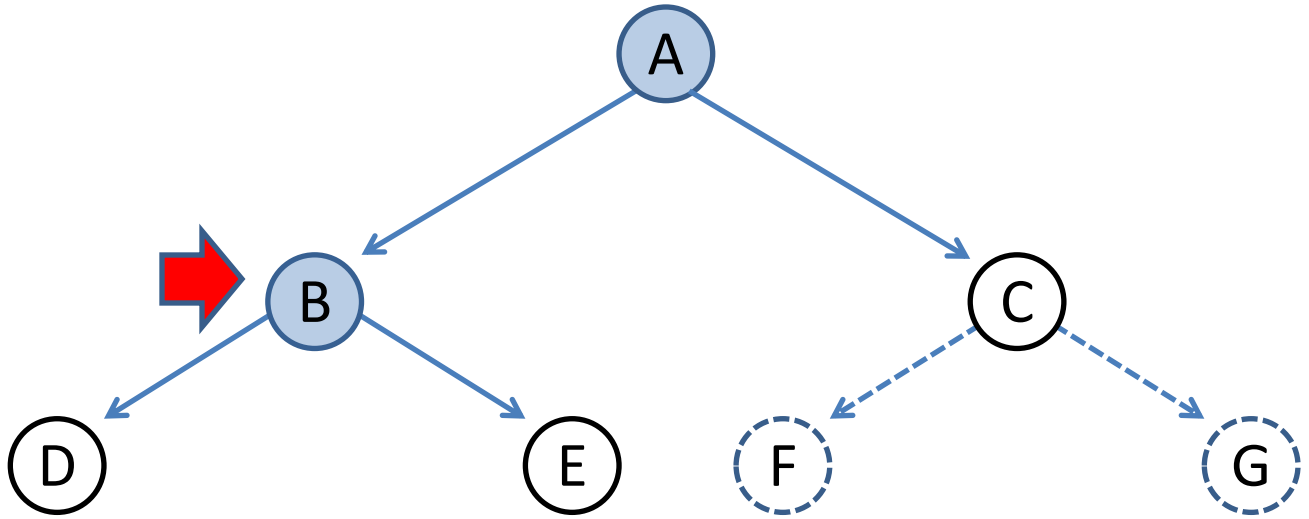
Breadth-first search

- Expand the shallowest unexpanded state
  - Depth : how many actions were made to generate the corresponding state
- Non-recursive implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



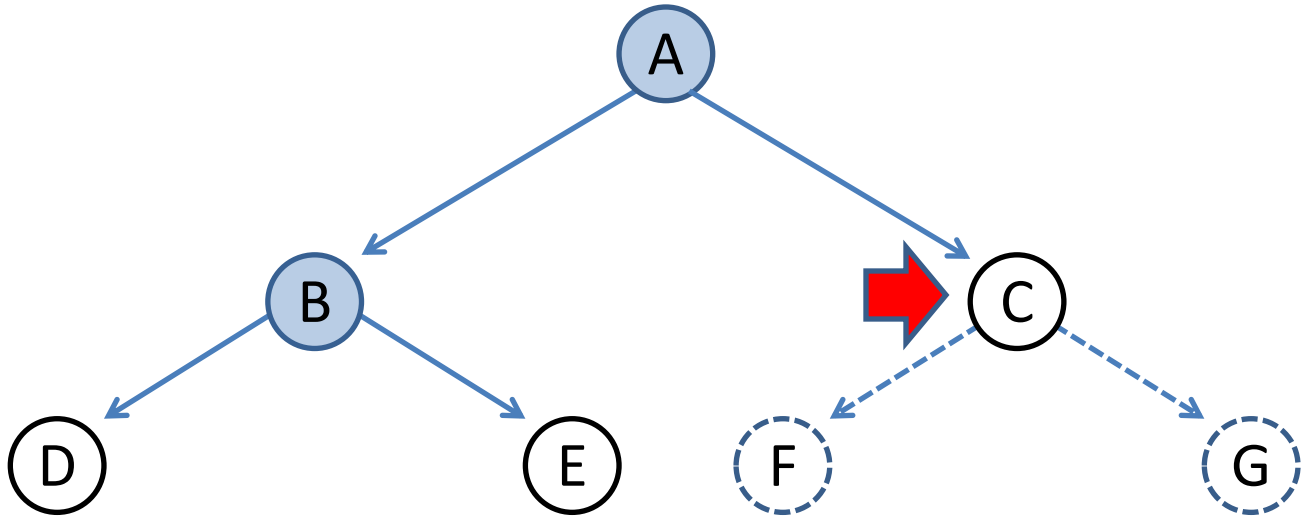
Breadth-first search

- Expand the shallowest unexpanded state
  - Depth : how many actions were made to generate the corresponding state
- Non-recursive implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



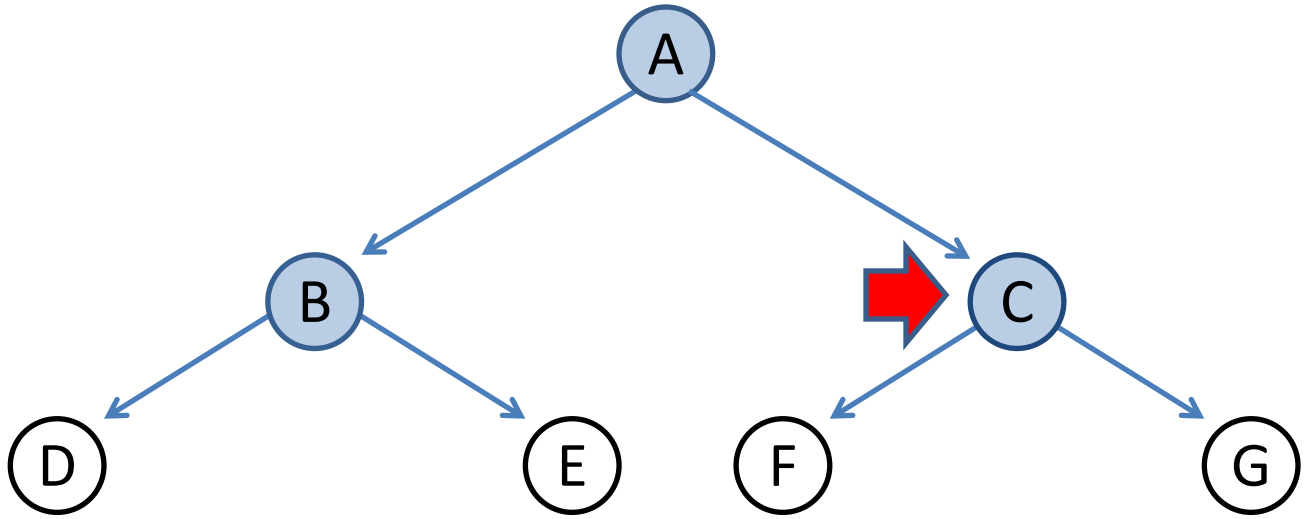
Breadth-first search

- Expand the shallowest unexpanded state
  - Depth : how many actions were made to generate the corresponding state
- Non-recursive implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

- Expand the shallowest unexpanded state
  - Depth : how many actions were made to generate the corresponding state
- Non-recursive implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



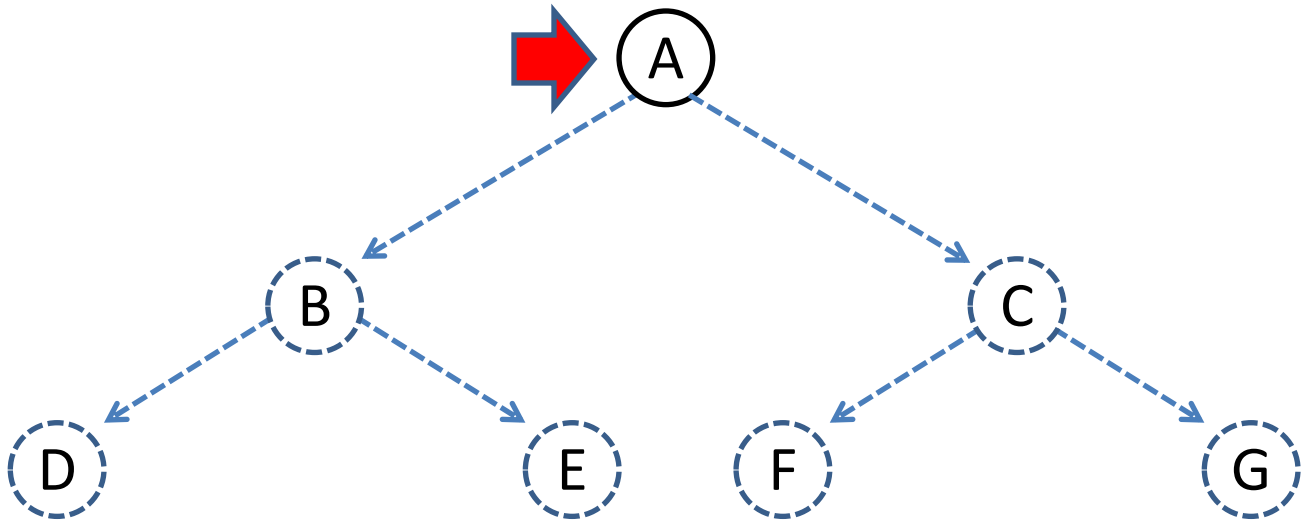
Pseudo-code for BFS

1. Generate an initially empty queue and call it OPEN.
   OPEN is the fringe.
2. Insert the start state into OPEN.
3. Dequeue a state n from OPEN. If dequeue fails because OPEN is empty, search fails (END).
4. If n is a goal state, the search succeeds (END).
5. Generate all successors to n and enqueue them into OPEN (end of OPEN).
6. Destroy node n
7. Return to step 3.

Nodes are structures that organize states. Each node contains all the information associated to a state (e.g., state itself, cost, and pointers to link it to the fringe, etc).

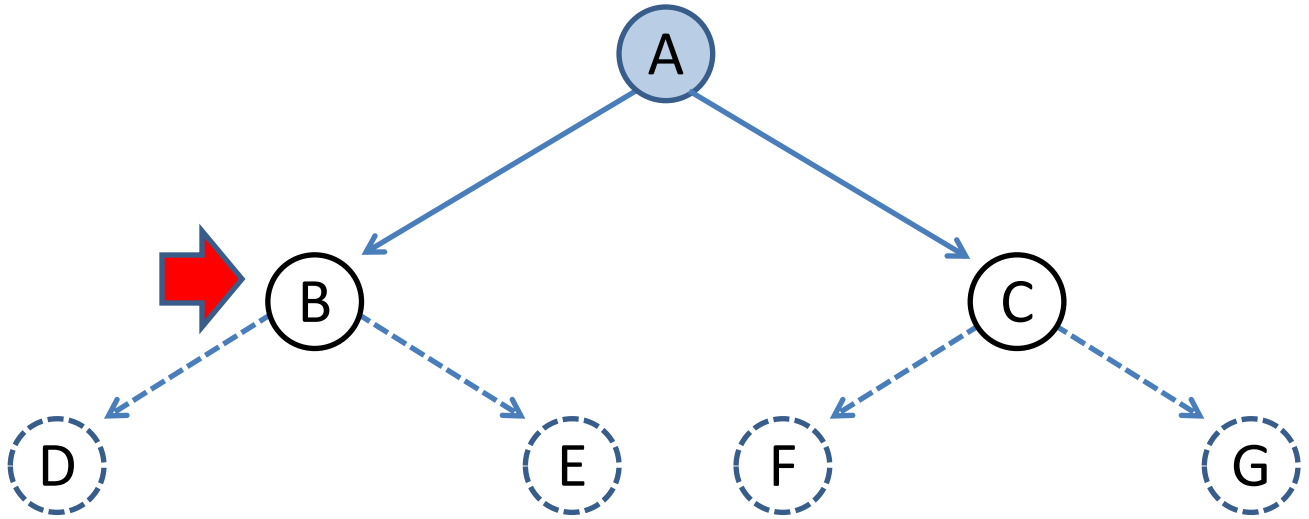
Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



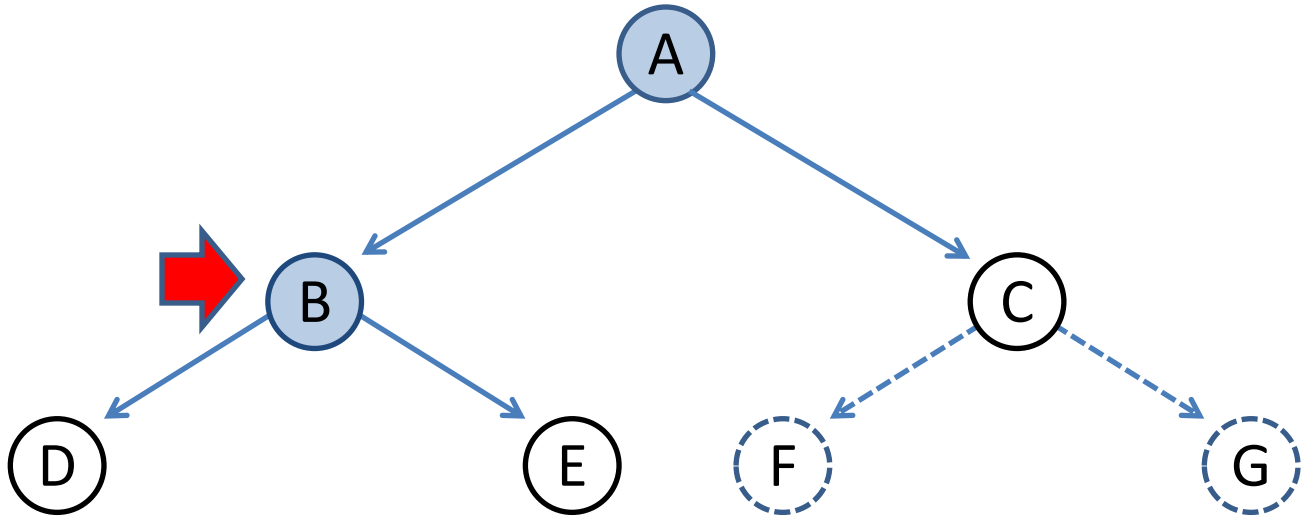
Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



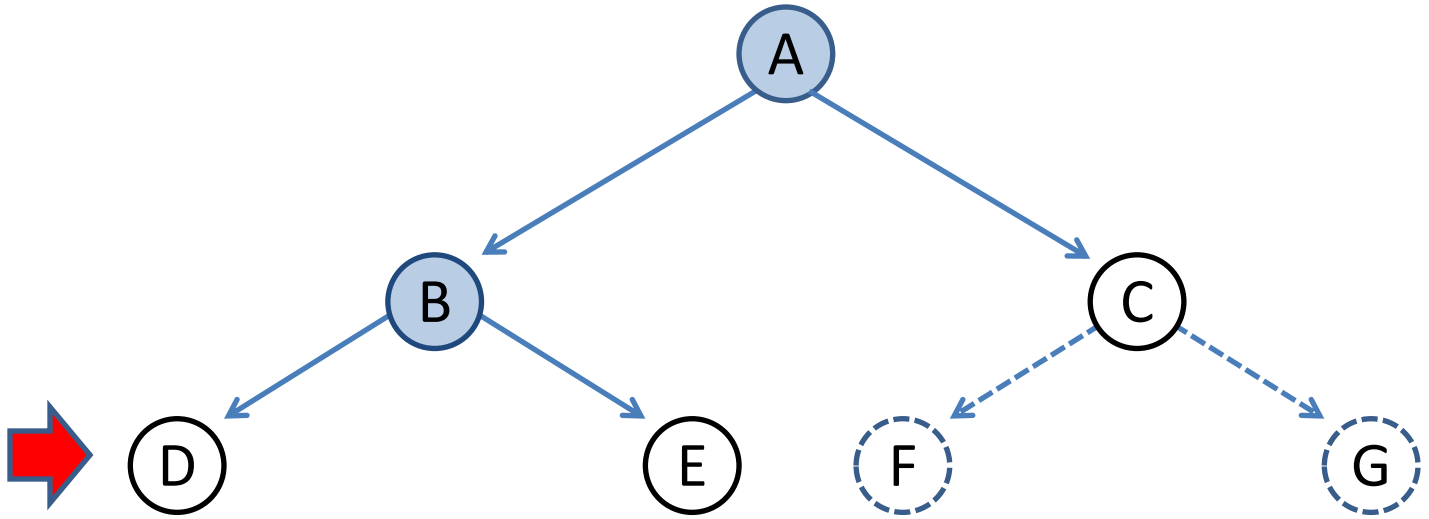
Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



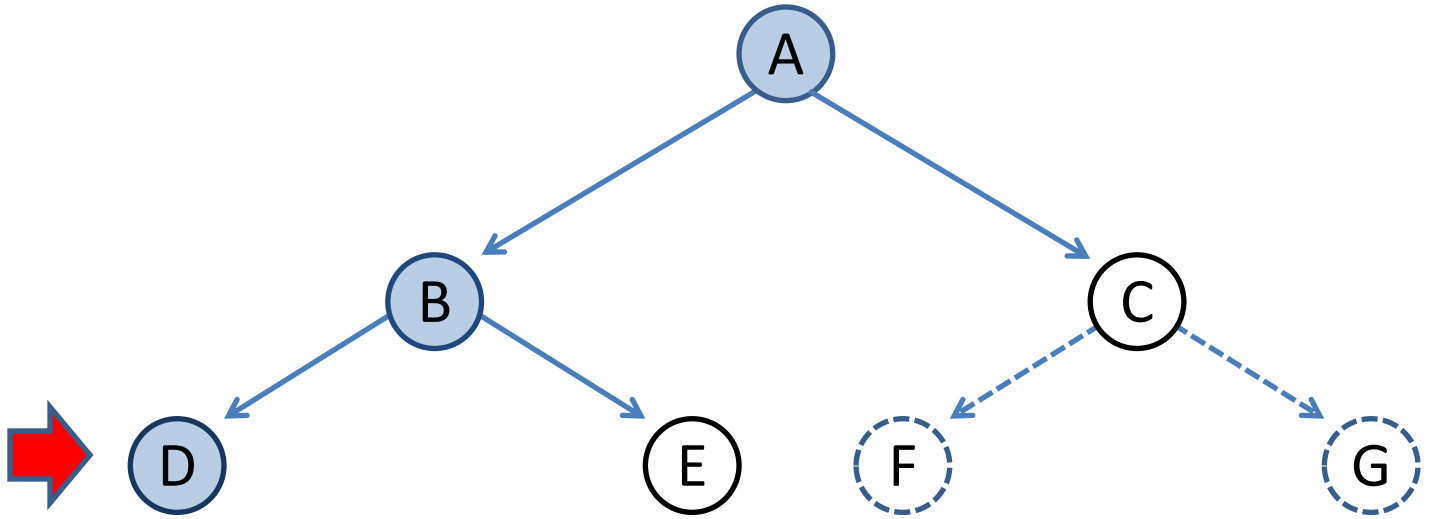
Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



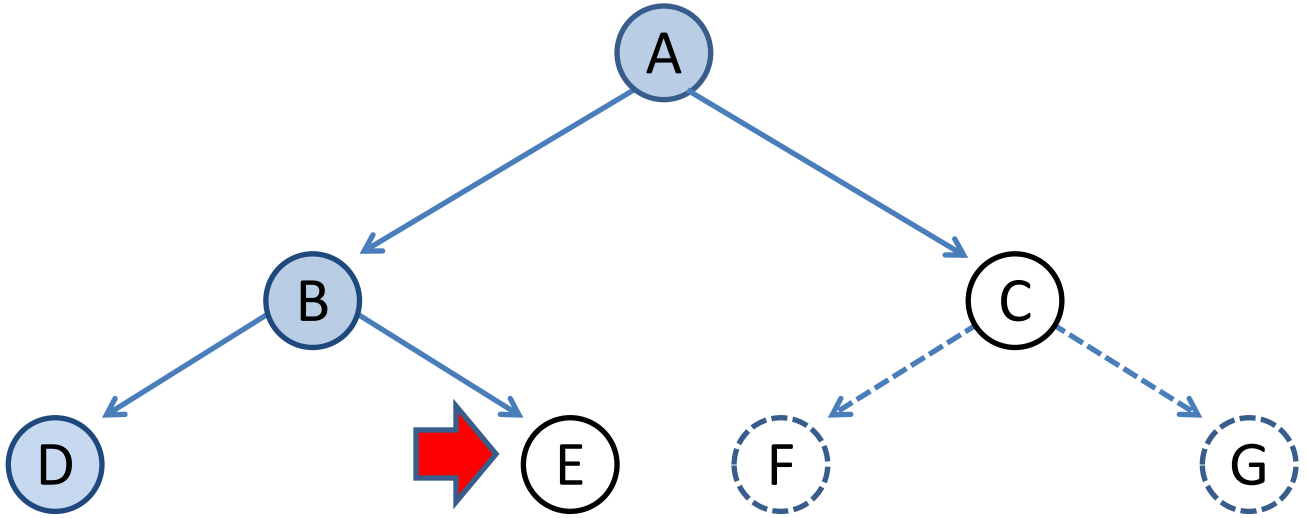
Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



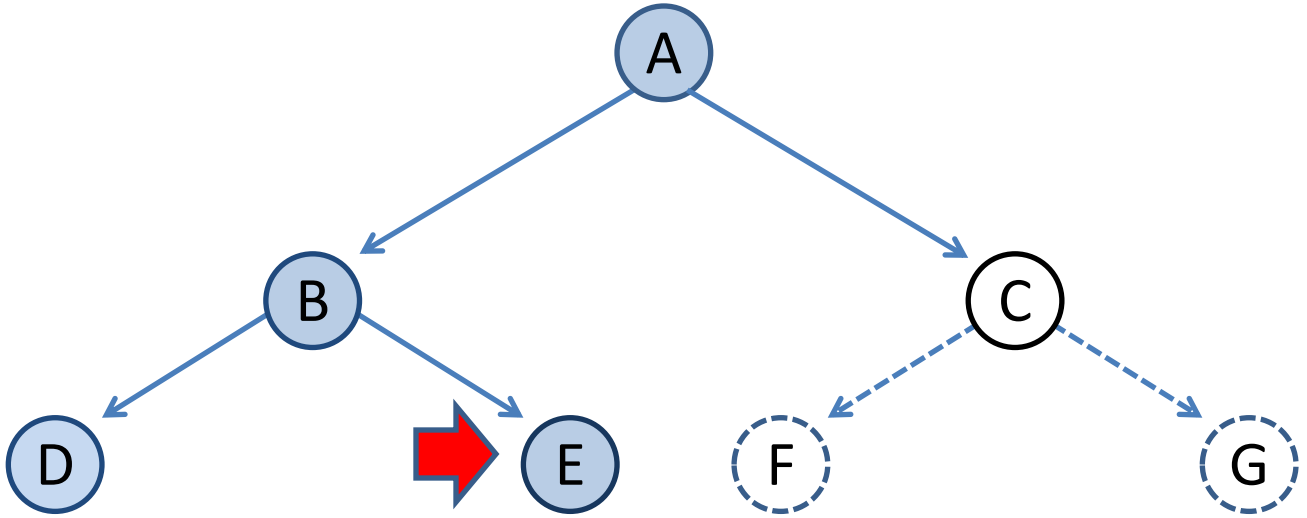
Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



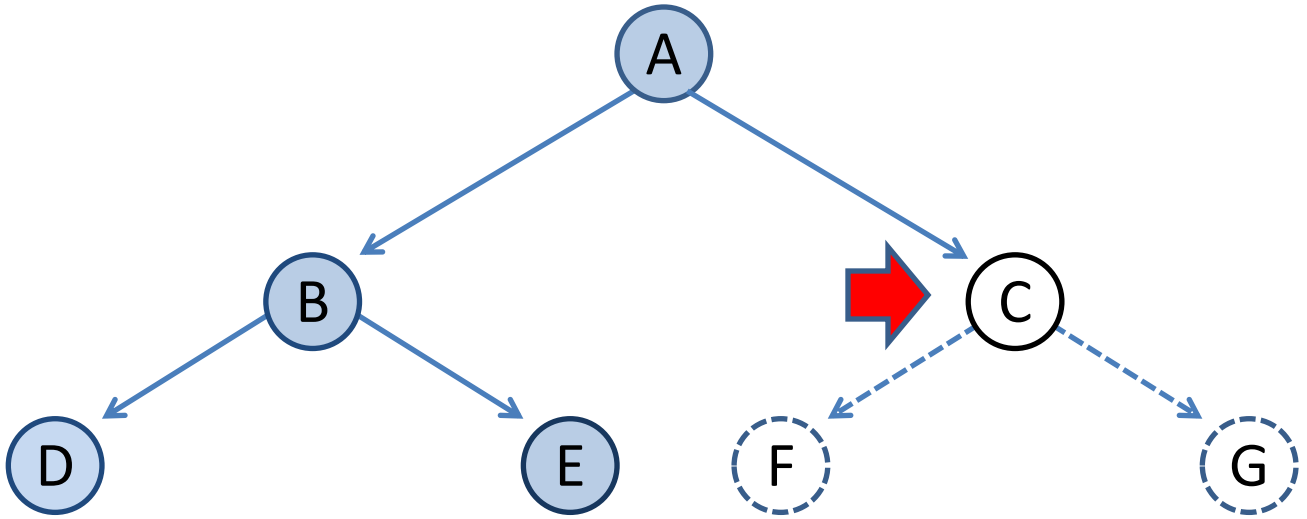
Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



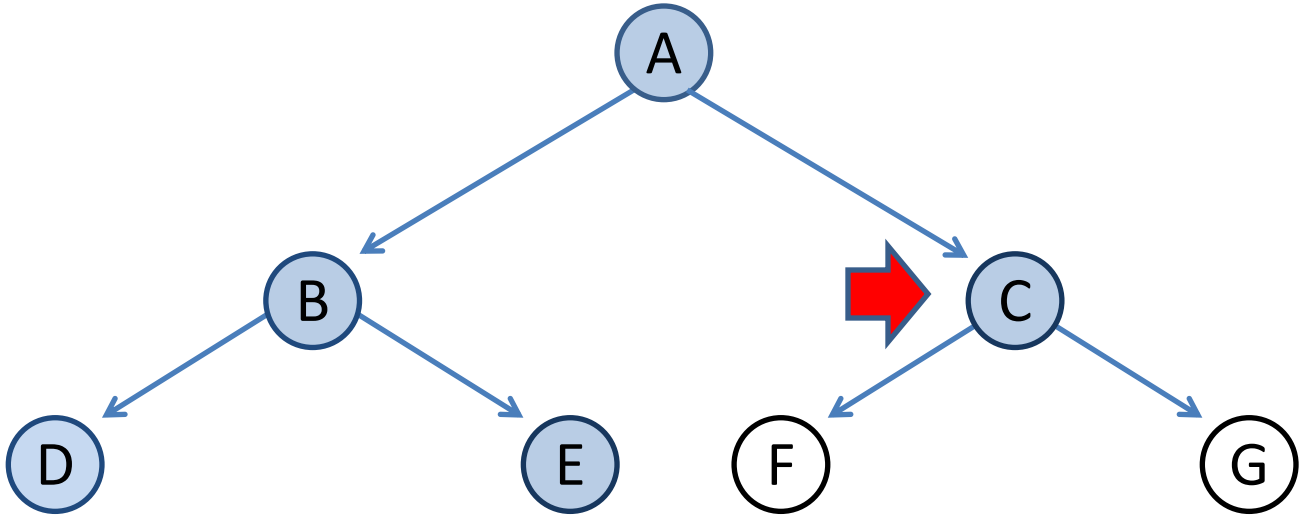
Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand deepest unexpanded state
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



Pseudo-code for DFS

1. Generate an initially empty stack and call it OPEN.
2. Insert the start state into OPEN.
3. Pop a state n from OPEN. If pop fails because OPEN is empty, search fails (END).
4. If n is a goal state, the search succeeds (END).
5. Generate all successors to n and push them onto OPEN (beginning of OPEN).
6. Destroy node n.
7. Return to step 3.

BFS vs DFS

- BFS
  - can always find the best solution (shortest path)
    - Always check the shallowest state (i.e., state with the lowest path cost).
  - Use a lot of memory
- DFS
  - May not find the best solution
    - Always check the deepest state. An unchecked state in fringe may be goal state. It is not selected because it is not deepest, or it has lower path cost.
  - Use less memory than BFS
  - may fail in infinite-depth paths (e.g., moving a tile back and forth)

Basic idea on searching for a solution

- Start from the **start state**
- Try different possible **actions**, and obtain **new states**
- Try different possible actions on new states, and obtain even **more new states**
- **Keep trying** until we find the goal state
- Check what actions were made to change the start state into the goal state → **solution**

- **Searching** is actually the process of trying different actions and checking **new states**.

DFS with CLOSED list

1. Generate an initially empty stack and call it OPEN, and a list called CLOSED.
2. Insert the start state into OPEN.
3. Pop a state n from OPEN. If pop fails because OPEN is empty, search fails (END).
4. If n is a goal state, the search succeeds (END).
5. Generate all successors to n.
6. Remove the successors that are already in OPEN or CLOSED list
7. Push the remaining successors onto OPEN (beginning of OPEN).
8. Add n into CLOSED list
9. Return to step 3.

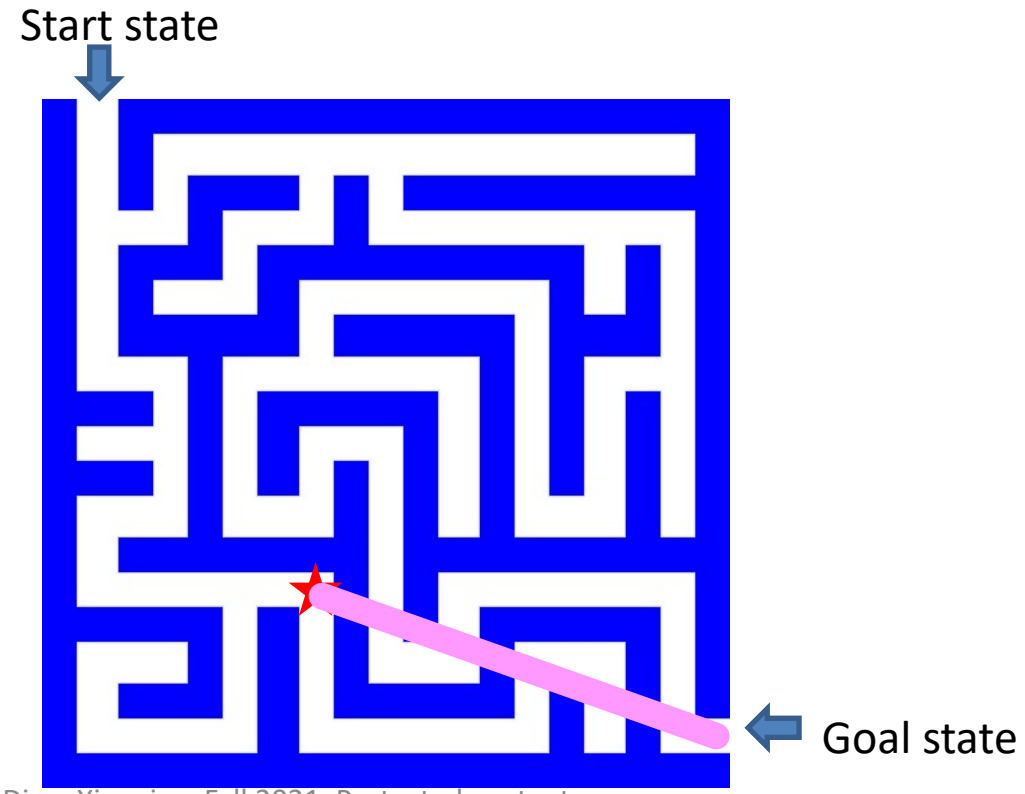
Informed search

- Idea: give the algorithm “hints” about the desirability of different states
  - Use an *evaluation function* to rank nodes and select the most promising one for expansion

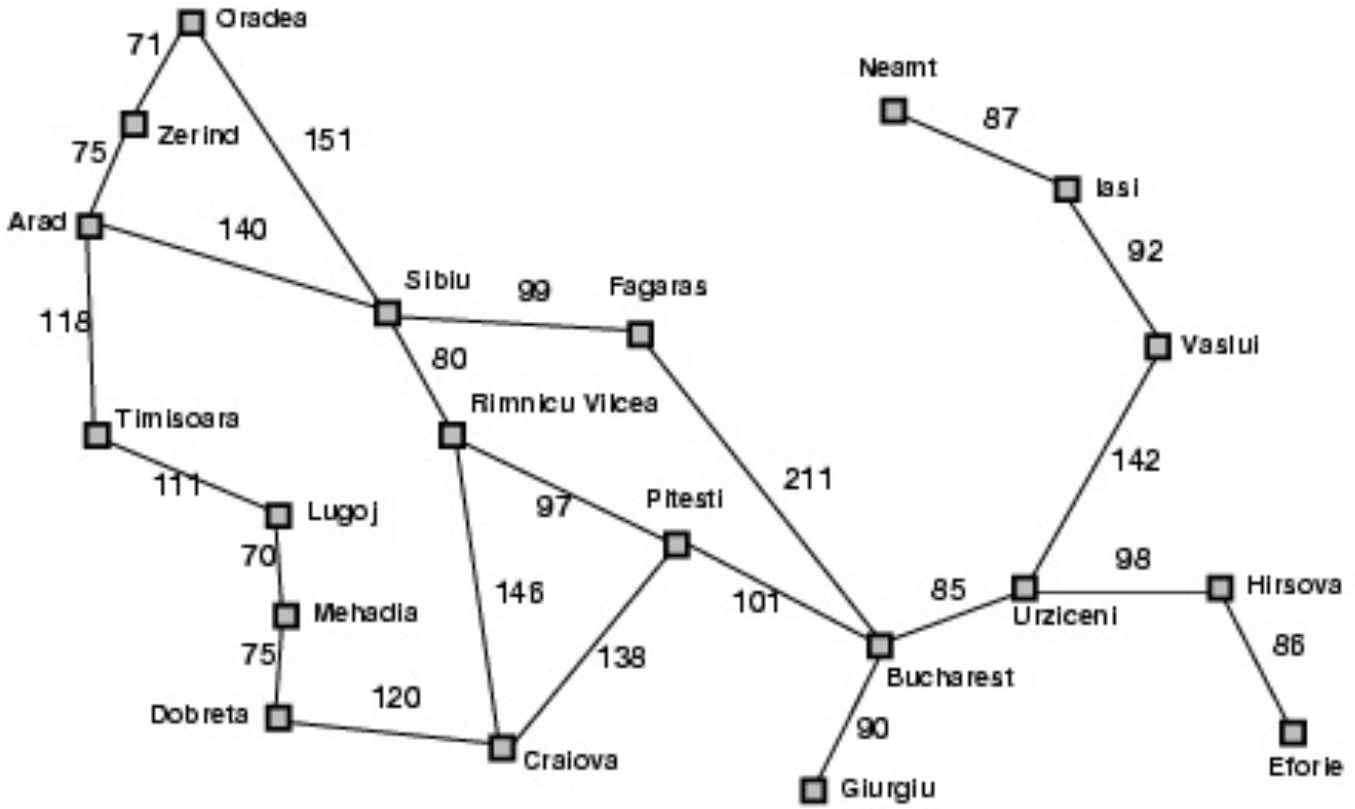
- Greedy best-first search
- A* search

Heuristic function

- Heuristic function $h(n)$ estimates the cost of reaching goal from node n
- Example:



Heuristic for the path finding problem

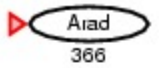


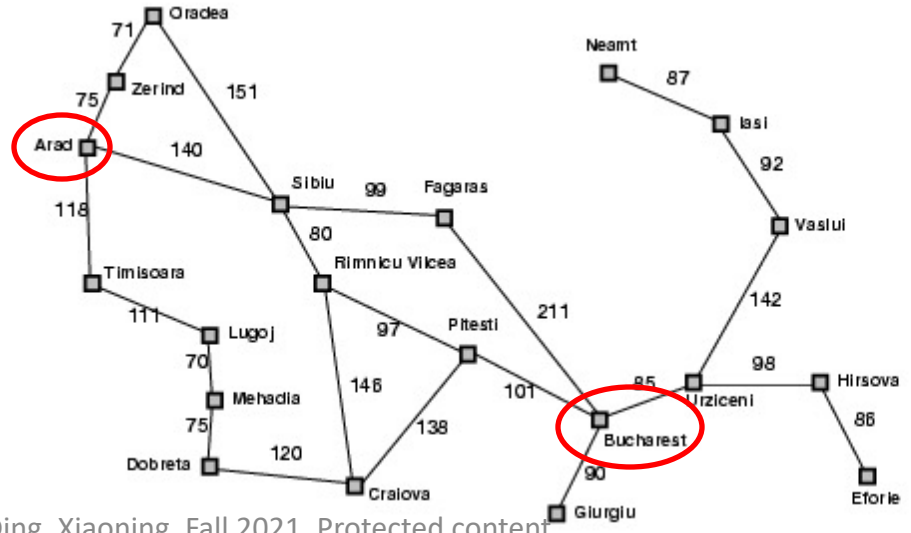
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

- Expand the node that has the lowest value of the heuristic function $h(n)$

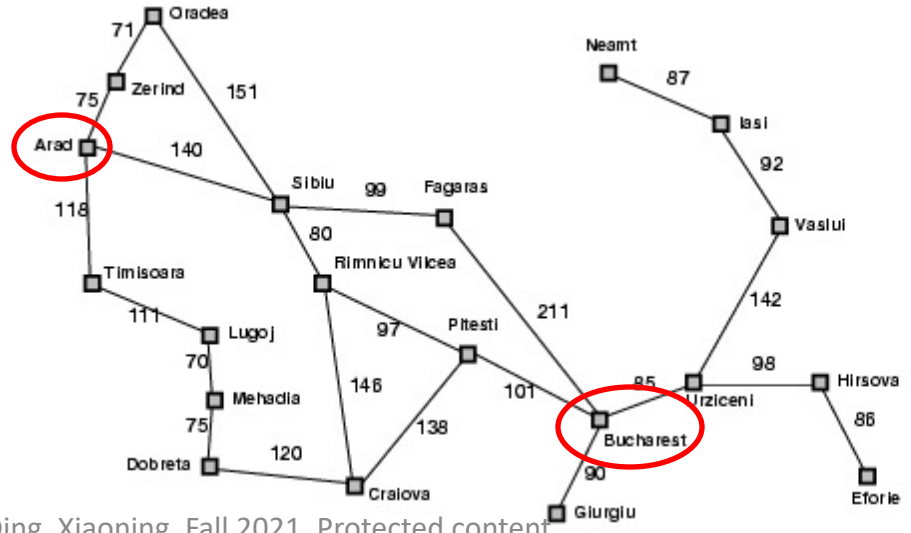
Greedy best-first search example





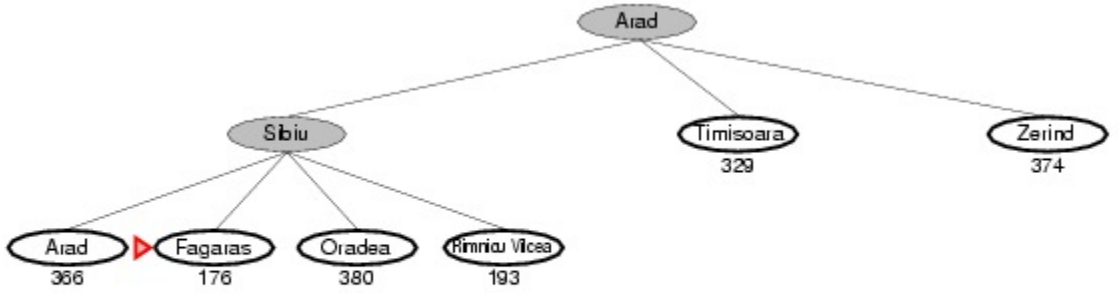
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

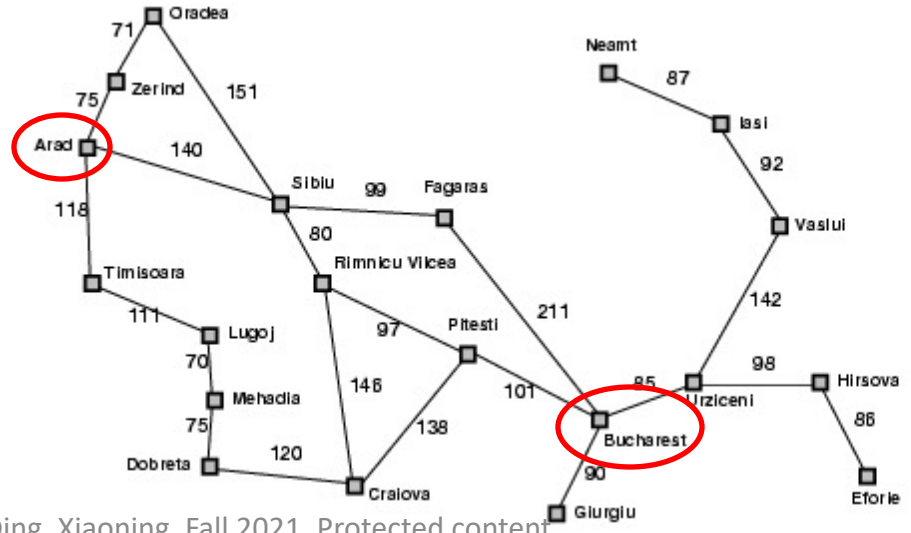
Greedy best-first search example



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

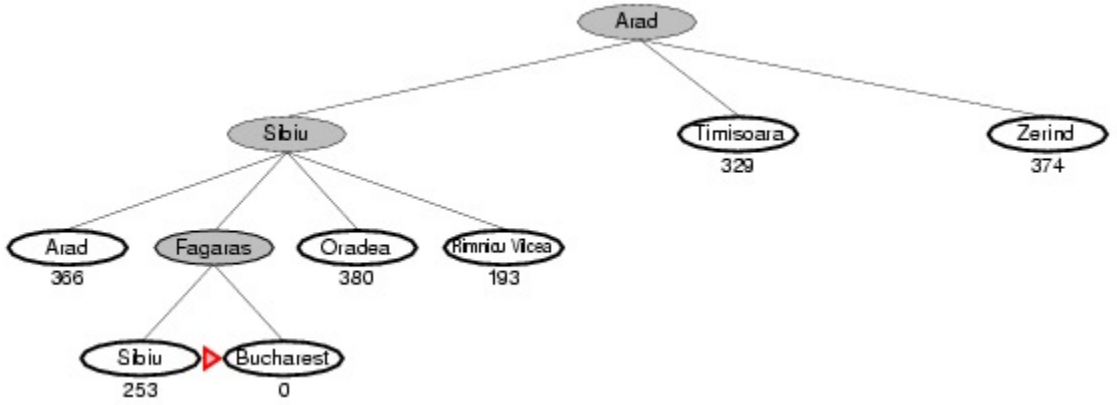
Greedy best-first search example

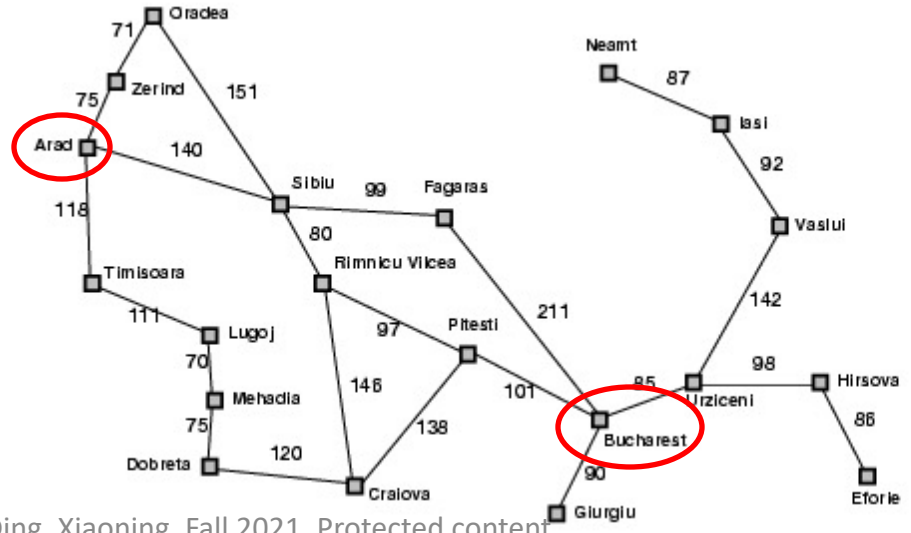




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

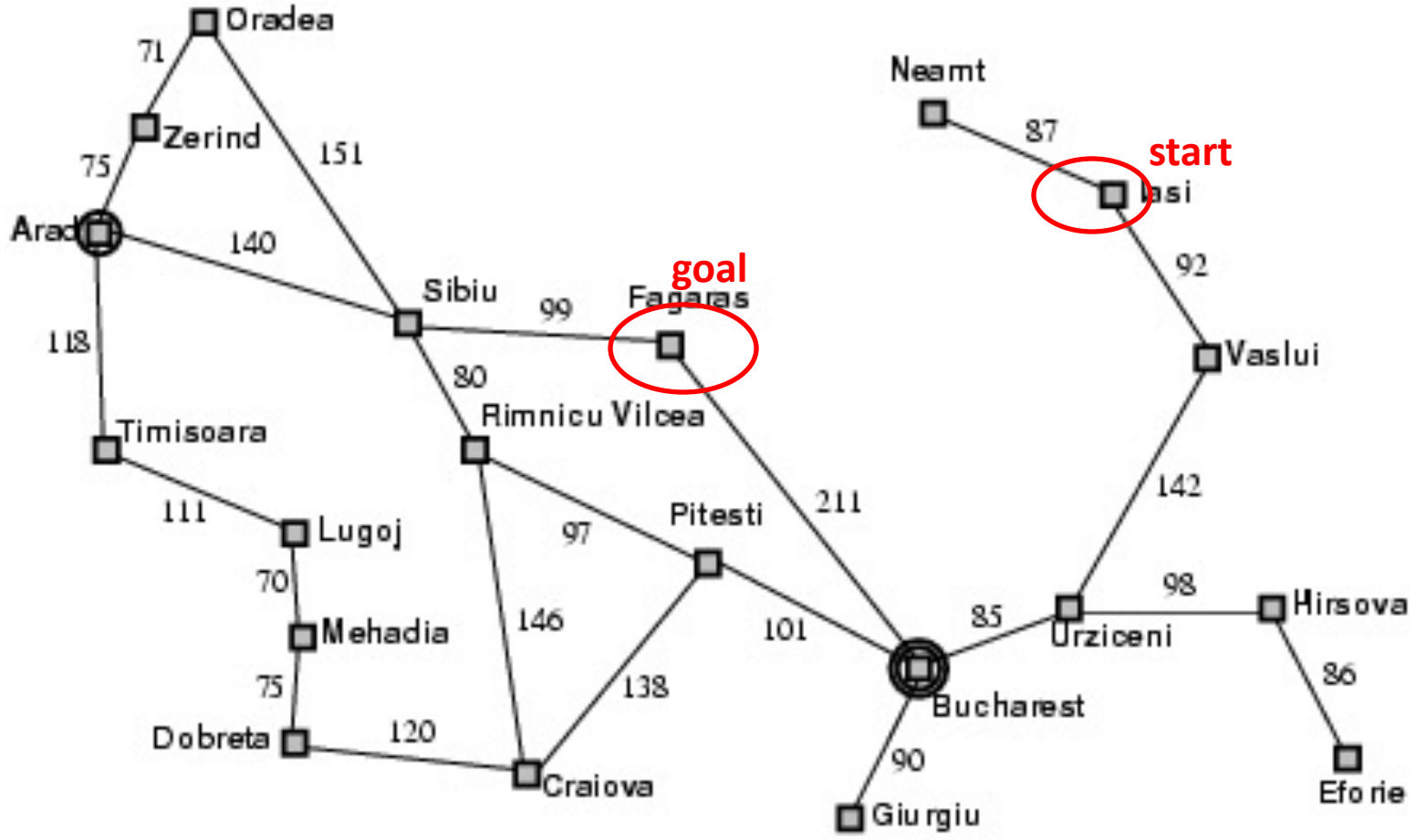
Greedy best-first search example



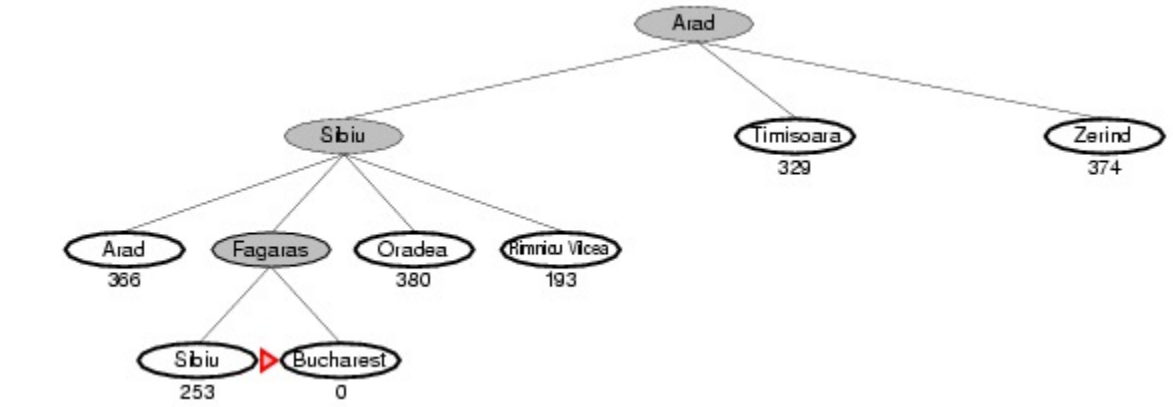


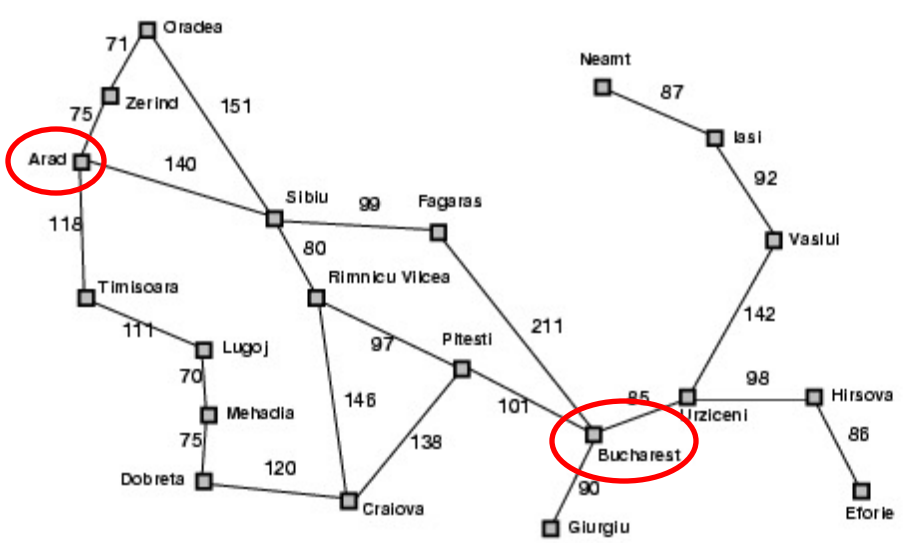
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search may get stuck in loops



Greedy best-first search is not optimal





Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search

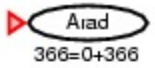
- Idea: avoid expanding paths that are already expensive
- The evaluation function $f(n)$ is the estimated total cost of the path through node n to the goal:

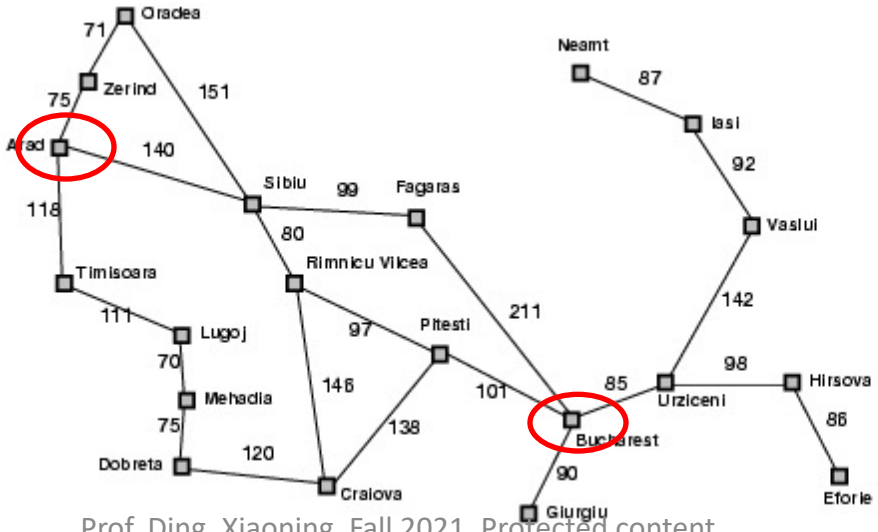
$$f(n) = g(n) + h(n)$$

$g(n)$: cost so far to reach n (path cost)

$h(n)$: estimated cost from n to goal (heuristic)

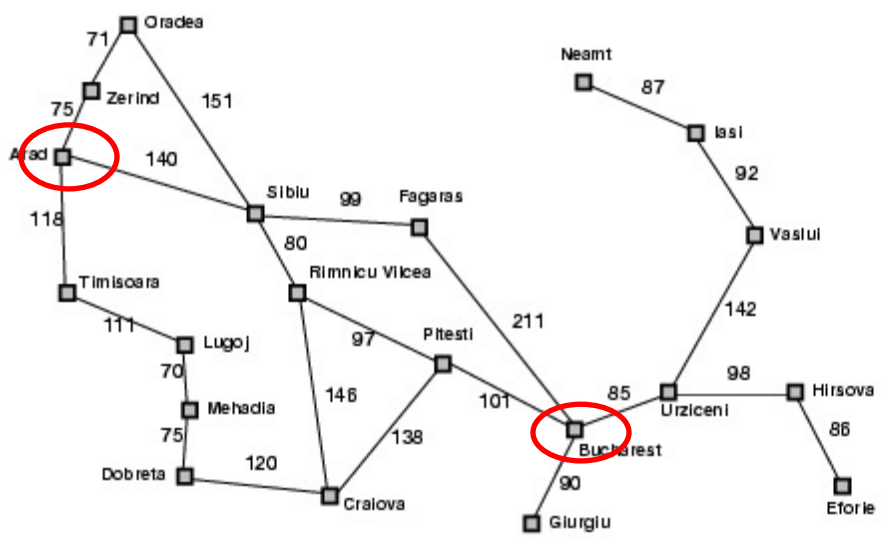
A* search example





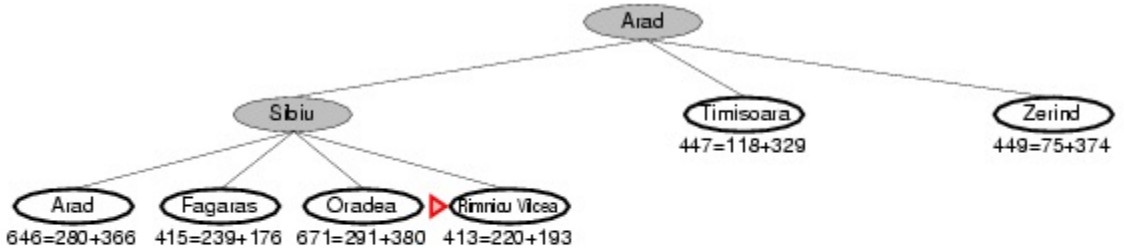
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

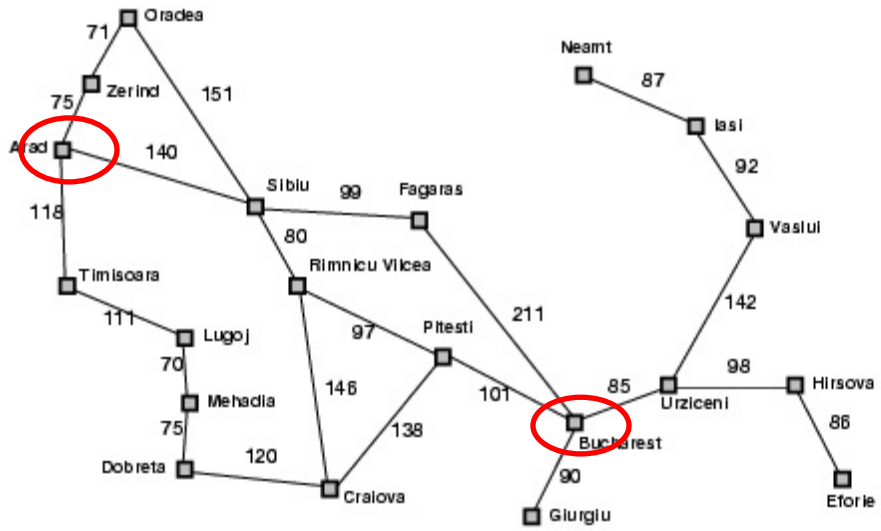
A* search example



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

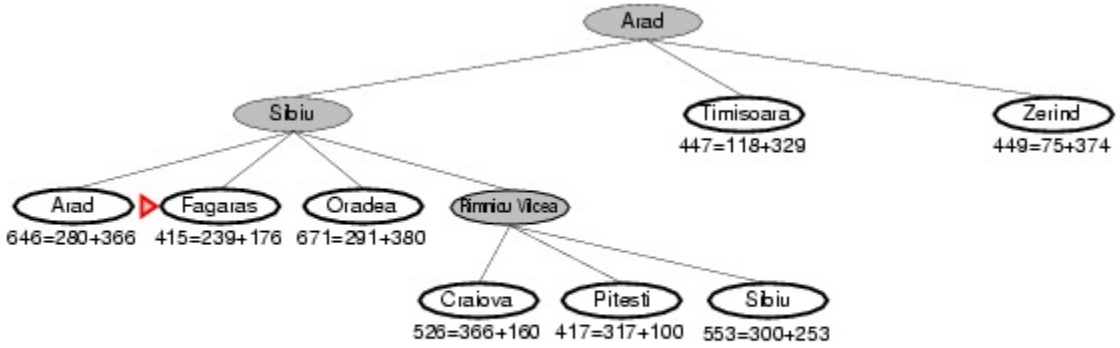


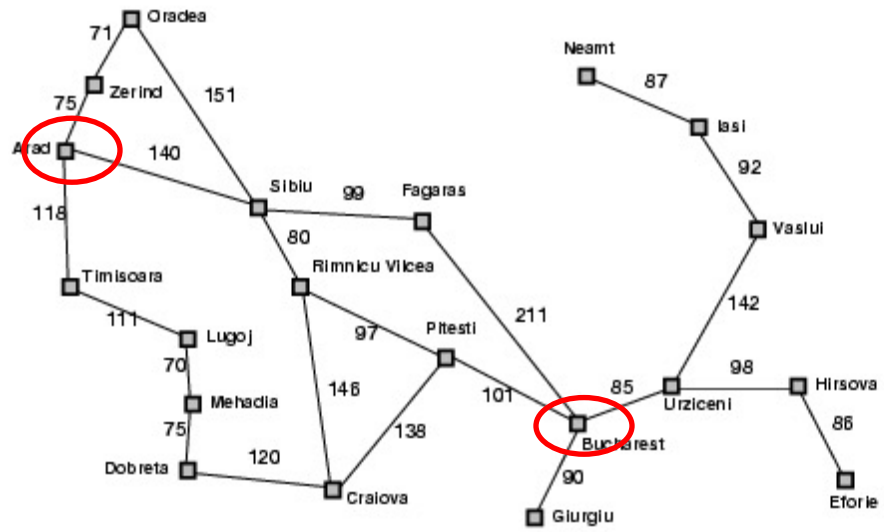


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

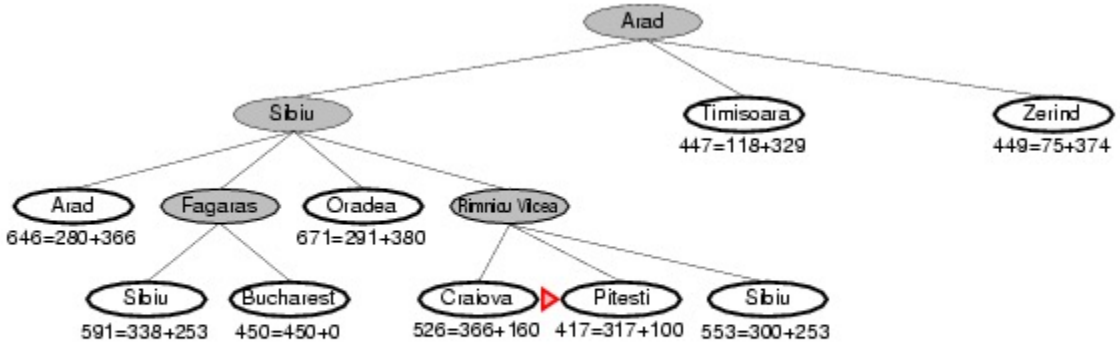


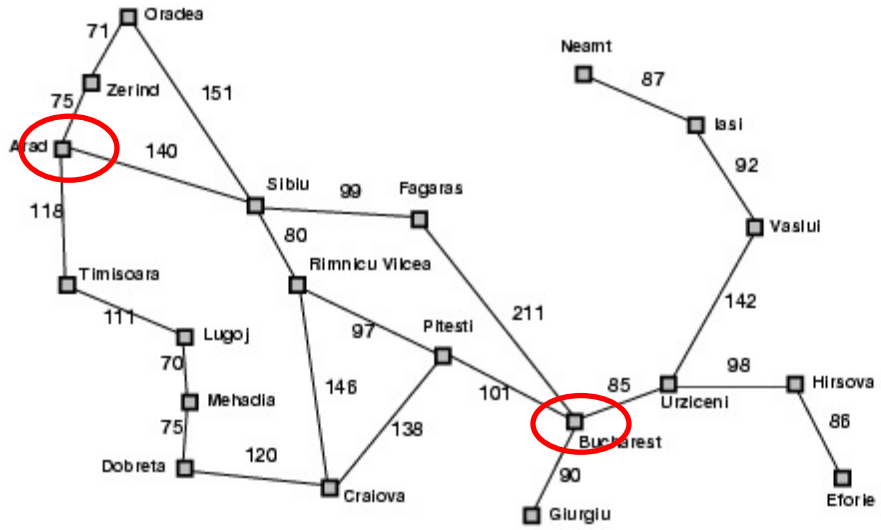


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

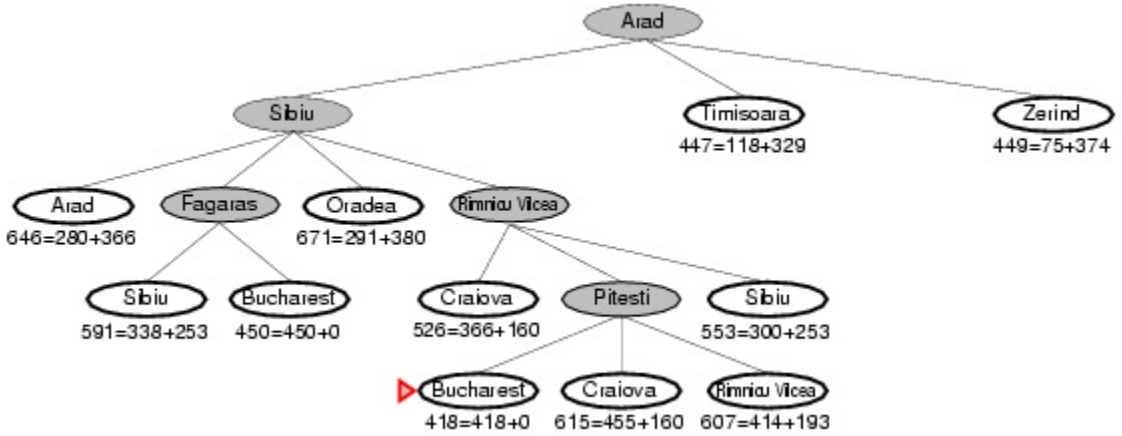


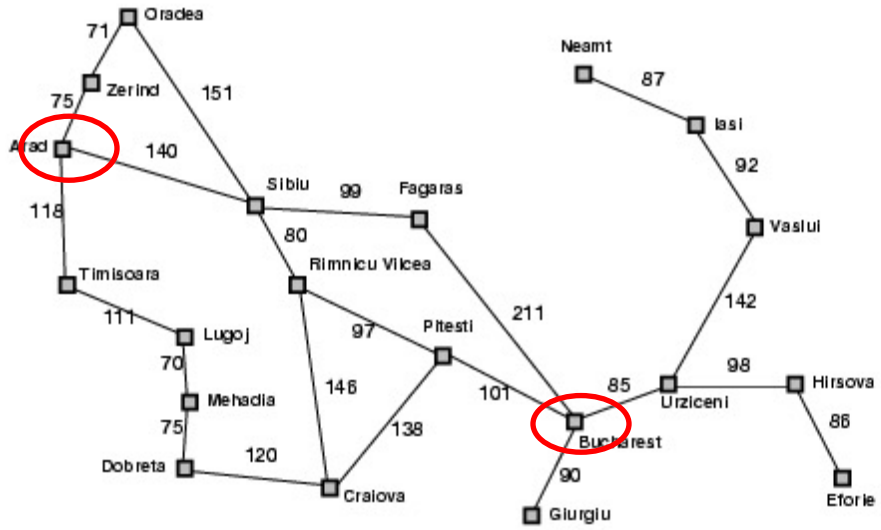


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example





Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Pseudo-code for A*

1. Generate an initially empty **priority** queue and call it OPEN, prioritized by f(n) value.
2. Generate an initially empty collection of states and call it CLOSED.
3. Insert the start state (or states) into OPEN.
4. Remove the state with smallest f(n) value (state n) from OPEN. If removal fails because OPEN is empty, search fails (END).
5. If n is a goal state, finish the search.
6. Generate all successors to n;
7. Remove and free the successors that are already in OPEN or CLOSED list.
8. Push remaining successors onto OPEN (prioritized by f(n) value).
9. Add n to CLOSED.
10. Return to step 4.

Admissible heuristics

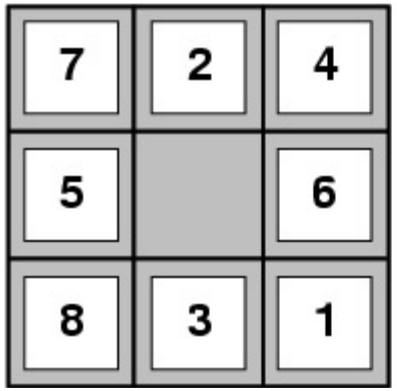
- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: straight line distance never overestimates the actual road distance
- Theorem: If $h(n)$ is admissible, A* is optimal

Designing heuristic functions

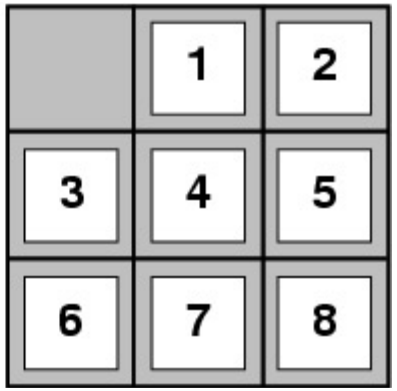
- Heuristics for the 8-puzzle

  $h_1(n)$ = number of misplaced tiles

  $h_2(n)$ = total Manhattan distance (number of squares from desired location of each tile)



State being evaluated



Goal State

$h_1(\text{start}) = 8$

$h_2(\text{start}) = 3+1+2+2+2+3+3+2 = 18$

- Are h_1 and h_2 admissible?

15-puzzle problem