Informática

Unidad 4: Algoritmos, y aplicaciones avanzadas

Ingeniería en Mecatrónica

Facultad de Ingeniería
Universidad Nacional de Cuyo

Dr. Ing. Martín G. Marchetta
mmarchetta@fing.uncu.edu.ar

UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERIA
en acción continua...

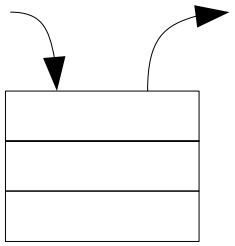
4A – Estructuras de datos avanzadas

4A – Estructuras de datos avanzadas

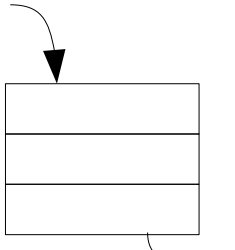
- Listas enlazadas
  - Son estructuras de datos compuestas por ***nodos***, cada uno de los cuales se relaciona con otros nodos a través de **apuntadores**
  - A diferencia de otras estructuras de datos, las listas enlazadas pueden **crecer/reducrise de a un elemento** a la vez cuando se lo necesita, y tienen poco *overhead* de procesamiento cuando crecen o se reducen
  - Existen 2 tipos:
    - Listas con enlace **simple**
    - Listas con enlace **múltiple** (normalmente doble)

4A – Estructuras de datos avanzadas

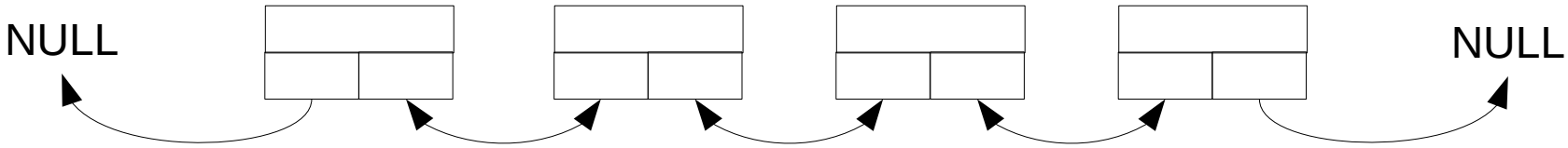
- Listas enlazadas
  - Listas con enlace simple:
    - Cada nodo apunta solamente al nodo “siguiente” (o al “anterior”)
  - Listas con enlace doble:
    - Cada nodo apunta al siguiente **y** al anterior
  - La listas enlazadas permiten implementar algunas estructuras de datos conocidas. Ej:
    - Pilas: LIFO (Last Input First Output)



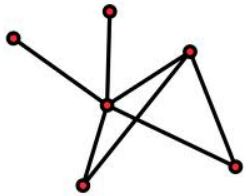
    - Colas: FIFO (First Input First Output)

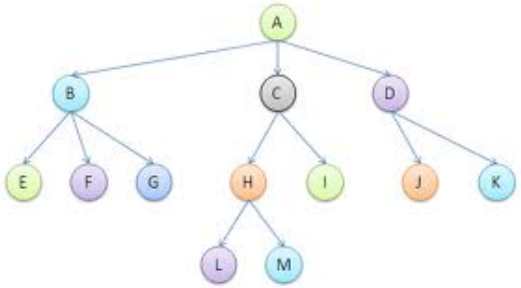


- Listas enlazadas
  - Listas que pueden recorrerse en ambos sentidos



- Grafos
  - Un grafo es una colección de **vértices** y **aristas** que conectan un subconjunto de estos vértices entre sí
  - Los grafos tienen muchas aplicaciones en inteligencia artificial e investigación operativa
  - Un tipo especial de grafo es el **árbol**
    - Un árbol es un grafo no dirigido, acíclico
    - Los árboles pueden tener un nodo especial, denominado ***raíz***, que no tiene padres; y pueden tener nodos ***hoja***, que no tienen hijos





4A – Aplicaciones avanzadas

- Listas enlazadas: Implementación
  - Cada nodo de la lista enlazada debe tener, al menos, 2 elementos asociados: un valor, y un apuntador al nodo siguiente (3 elementos si es una lista con enlace doble).
  - Por lo tanto, los nodos de la lista son struct

```
struct NODO_SUCESOR{
    int fila;
    int columna;
    struct NODO_SUCESOR *siguiente
};
```

  - Inicialización de la lista

```
struct NODO_SUCESOR *sucesores;
sucesores = malloc(sizeof(struct NODO_SUCESOR));
sucesores->fila = sucesores->columna = -1;
sucesores->siguiente = NULL;
```

4A – Aplicaciones avanzadas

- Al agregar un nodo a la lista, se debe: (a) crear el nodo; (b) Actualizar los apuntadores. Ejemplo:

```
struct NODO_SUCESOR *sucesores;  //Apuntador al inicio de la lista
struct NODO_SUCESOR *sucesor_actual; //Nodo actual
...
sucesor_actual = sucesores;                  //Buscamos el ultimo nodo
while(sucesor_actual->siguiente != NULL) //partiendo desde el 1°
    sucesor_actual = sucesor_actual->siguiente;
sucesor_actual->siguiente = malloc(sizeof(struct NODO_SUCESOR));
sucesor_actual->siguiente->fila = -1;
sucesor_actual->siguiente->columna = -1;
sucesor_actual->siguiente->siguiente = NULL;
```

4A – Aplicaciones avanzadas

- Al eliminar un nodo a la lista, se debe: (a) Actualizar los apuntadores; (b) liberar la memoria. Ejemplo:

```
struct NODO_SUCESOR *temp; //Apuntador auxiliar

…

//Tenemos los nodos i, i+1 e i+2, y queremos eliminar el nodo i+1

//Guardamos un apuntador auxiliar al nodo i+2 (para no “perderlo”)

temp = sucesor_actual->siguiente->siguiente;

//Liberamos la memoria del nodo i+1

free(sucesor_actual->siguiente);

//Asociamos el nodo i+2 (apuntado por temp) al nodo i

sucesor_actual->siguiente = temp;
```

4A – Aplicaciones avanzadas

- Listas enlazadas: implementación
  - Si la lista tiene enlaces múltiples, se deben actualizar todos los apuntadroes relevantes. El no hacerlo puede causar:
    - Violaciones de segmento: si un apuntador queda apuntando a una dirección de memoria liberada
    - Que se pierda el apuntador a uno o más nodos, con lo cual habrá memoria no liberada que no será accesible
      - A esto se le llama ***fuga de memoria*** (***memory leak***), dado que hay memoria que no puede ser asignada por el Sistema Operativo a otro proceso, pero que tampoco es usada por el proceso que la reservó

4B – Búsqueda y ordenamiento

4B – Búsqueda y ordenamiento

- Algoritmos de ordenamiento
  - Bubblesort: pseudocódigo

```
hacer
    para i=1 hasta n-1
        si v[i-1] > v[i] entonces
            intercambiar v[i-1] y v[i]
        fin si
    fin para
mientras hayan substituciones
```

4B – Búsqueda y ordenamiento

- Algoritmos de ordenamiento
  - Quicksort: El algoritmo elige un pivote y mueve todos los elementos menores que él a su izquierda, y el resto a su derecha, y luego repite el proceso para la mitad izquierda, y luego para la mitad derecha

```
funcion quicksort(v, izq_idx, der_idx, pivote_idx)
  valor_pivote = v[pivote_idx]
  i = izq_idx
  j = der_idx
  mientras i <= j
    mientras v[i] < valor_pivote
      i += 1
    fin mientras
    mientras v[j] > valor_pivote
      j -= 1
    fin mientras
    si i <= j
      intercambiar v[i] con v[j]
      i += 1
      j -= 1
    fin si
  fin mientras
    quicksort(v, izq_idx, j, (izq_idx + j - 1)/2)
    quicksort(v, j+1, der_idx, (j + 1 + der_idx)/2)
fin quicksort
```

4B – Búsqueda y ordenamiento

- Algoritmos de búsqueda
  - Búsqueda binaria
    - Pre-condiciones: el arreglo debe estar ordenado (asumimos orden de menor a mayor)
    - Post-condiciones: se identifica el subíndice en el que se encuentra el elemento buscado
    - Procedimiento:
      - La idea es tomar el elemento que está justo a la mitad del arreglo (pivote), y compararlo con el elemento buscado.
      - Si el elemento buscado es menor que el pivote, repetimos el proceso, pero en lugar de tomar el arreglo completo, tomamos la mitad del arreglo que está a la izquierda del elemento pivote.
      - Este procedimiento se repite, reduciendo el arreglo utilizado en la búsqueda en cada intento a la mitad izquierda o derecha, dependiendo de la comparación del elemento buscado con el pivote, hasta que se encuentra el elemento o se determina que el elemento no está en el arreglo.

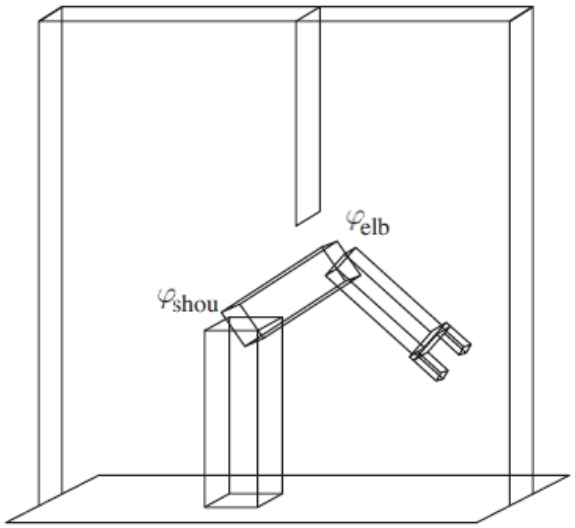
4B – Búsqueda y ordenamiento

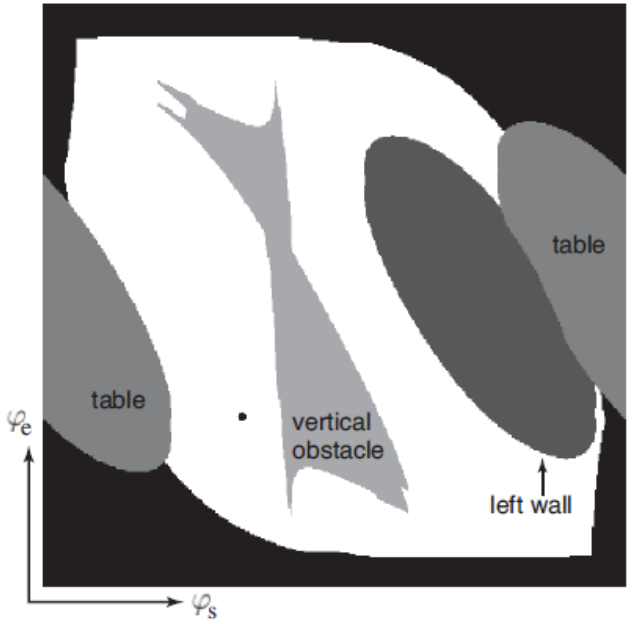
- Algoritmos de búsqueda
  - Búsqueda binaria
    - Consideraciones:
      - El procedimiento es conceptualmente similar al del algoritmo quicksort, y tiene una complejidad algorítmica similar O(n log n)
      - Se suele implementar mediante recursión, puesto que es más compacto y fácil de depurar. Sin embargo, si se buscará en arreglos muy grandes, puede implementarse también iterativamente.

4C – Aplicaciones

4C – Aplicaciones

- Ejemplo de aplicación de árboles: búsqueda de rutas en espacios discreto

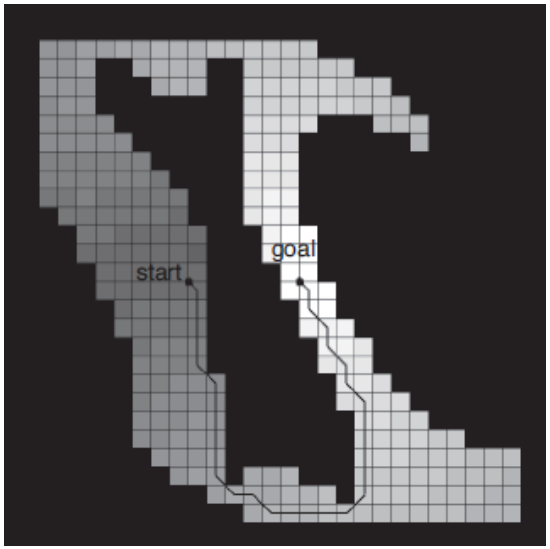




- Espacio físico vs. Espacio de búsqueda

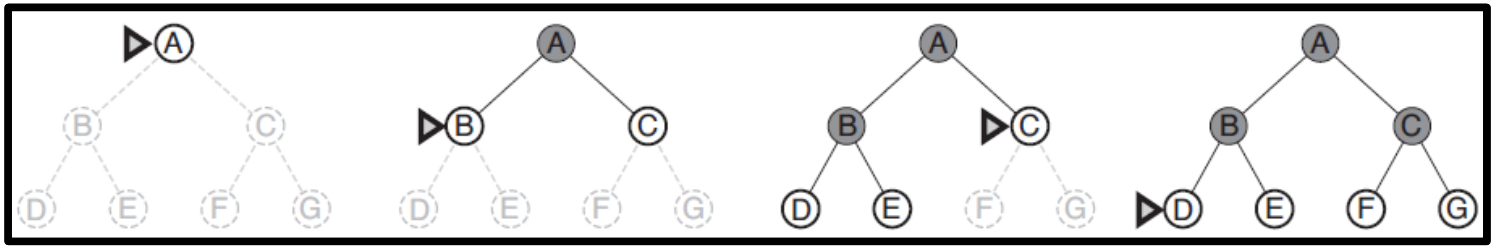
4C – Aplicaciones

- Ejemplo de aplicación de árboles: búsqueda de rutas en espacios discreto



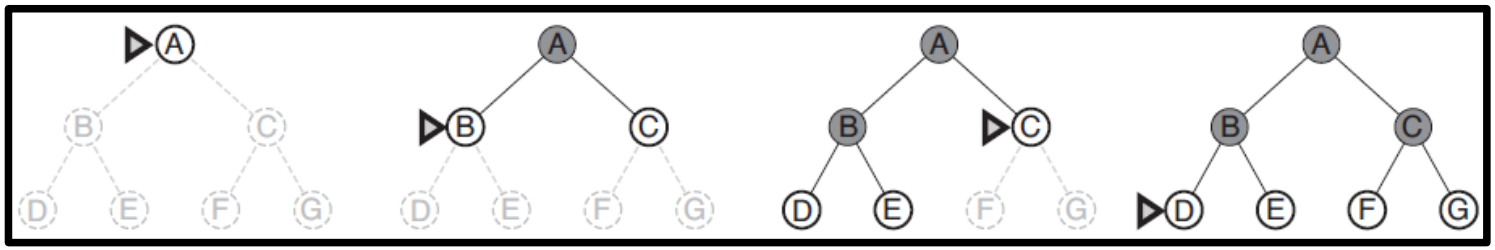
Espacio de búsqueda

Árbol de búsqueda



4C – Aplicaciones

- Backtracking
  - Consiste en retornar al punto de decisión anterior cuando la decisión actual se han explorado totalmente
  - Permite implementar una ***búsqueda exhaustiva***
  - Si se combina con validaciones para evitar lazos, es un algoritmo ***completo***
  - Puede combinarse con cualquier heurística o estrategia de exploración



4C – Aplicaciones

- Algoritmo avaro
  - Es un algoritmo que adopta el enfoque de elegir, en cada punto de decisión, la opción localmente óptima
  - No garantiza encontrar el óptimo global
  - En general, obtiene soluciones de buena calidad muy rápido
  - Dependiendo de la calidad de la función con la que se calcula la opción “localmente óptima”, el algoritmo puede encontrar “calles sin salida” requiriendo mucho backtracking
- Programación dinámica
  - Consiste en resolver un problema subdividiéndolo en problemas más chicos y resolviendo estos problemas ***sólo una vez,*** y ***almacenando*** estas ***soluciones intermedias***

4C – Aplicaciones

- Programación dinámica (cont.)
  - Requiere que se cumpla que:
    - Subestructura óptima: la solución óptima de un problema puede obtenerse como una combinación de soluciones óptimas de sus sub-problemas
    - Problemas solapados: la división del problema en problemas más pequeños, y la división de estos en problemas más pequeños, etc. muestra un patrón donde se repiten estos problemas una y otra vez
  - En este contexto, los problemas “repetidos” se resuelven una sola vez y su solución se “memoriza”, evitando recalcularlas
  - Ejemplo:
    - Serie de Fibonacci (Fi = Fi-1 + Fi-2)
    - Encontrar el camino más corto entre 2 puntos