Estructuras de datos y librerías


folivares13@alumnos.utalca.cl

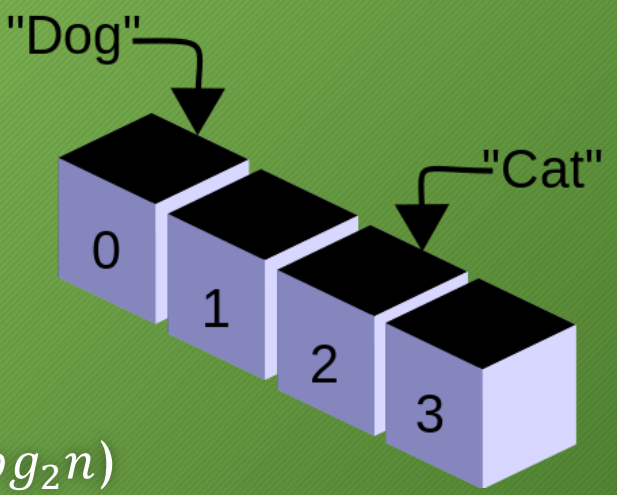

@Jestter

Lectura complementaria

- El contenido de esta presentación se encuentra en el **capítulo 2** del libro **Competitive Programming 3**

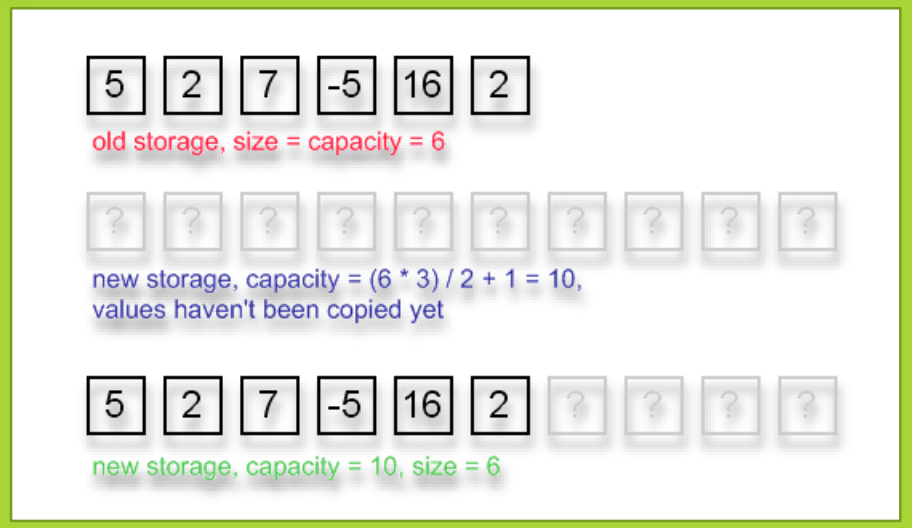
Static Array

- Es la estructura más común
- Permite almacenar una secuencia de datos y acceder a través de los índices
- No se puede redimensionar
- Operaciones típicas:
  - Acceder a los elementos a través de los índices en $O(1)$
  - Ordenar los elementos en ~~$O(n^2)$~~, $O(n\ log_2 n)$, $O(n)$
  - Hacer búsqueda lineal en el array en $O(n)$
  - Hacer búsqueda binaria en el array **si está ordenado** en $O(log_2 n)$



Dynamically-Resizeable Array

- Similar al array pero tiene la particularidad de poder cambiar su tamaño en tiempo de ejecución
- Se recomienda su uso cuando no se conoce el tamaño de la secuencia de elementos
- **vector** en C++, **ArrayList** en Java, **listas** en Python, etc.
- Operaciones típicas:
  - Las mismas que en Static Array
  - Agregar/eliminar un elemento al final del array en $O(1)$. Nota: no siempre es $O(1)$
  - Agregar/eliminar elementos del array en una posición intermedia en $O(n)$.
  - Eliminar todos los elementos del array



Sorting (visualizar)

- $O(n^2)$:
  - Bubble/Selection/Insertion Sort, etc.
  - Horriblemente lentos
  - No se recomienda su uso en competencias
- $O(n \log_2 n)$:
  - Merge/Heap/Quick Sort, etc.
  - La elección por defecto para competencias.
  - Implementación de alguno de estos ya viene por defecto en los lenguajes
  - **sort/stable_sort** en C++, **Collections.sort** en Java, **sorted (o list.sort)** en Python
- $O(n)$:
  - Counting/Radix/Bucket Sort, etc.
  - Raramente usados
  - Para casos en que los datos tienen ciertas características especificas
  - Generalmente no vienen implementados en los lenguajes (tienen que programarlos)

Search

- $O(n)$:
  - Recorrer todos los elementos en la secuencia hasta encontrar el buscado
  - Evitarlo si es posible
- $O(log_2 n)$:
  - Búsqueda binaria
  - **lower bound, upper bound o binary search** en C++. **Collections.binarySearch** en Java. Módulo **bisect (bisect_left, bisect_right)** en python
- $O(1)$:
  - Hashing
  - Útil cuando se requiere acceso realmente rápido a valores conocidos
  - Riesgo de colisiones (obtener valores incorrectos). Aunque no es un riesgo tan alto con una función de hash apropiada.
  - Para esto se utilizan estructuras como Hash Tables

Array of Booleans

- Si el array solo necesita contener booleans existe una estructura como alternativa al array.
- bitset en C++ y BitSet en Java
- La implementación de C++ está optimizada para el espacio, por lo general cada elemento utiliza un solo bit. El tamaño es fijo, si se requiere tamaño variable existe vector<bool>
- La implementación de Java funciona con tamaño variable. Se pueden efectuar operaciones AND, OR, XOR entre bitsets

Bitwise Operators

- Los operadores bitwise trabajan a nivel de bits sobre los valores.
- NOT (!): Negación lógica. Invierte los bits de un valor de 1 a 0 y de 0 a 1.
- AND (&): realiza el “and” lógico entre dos valores. 1 si ambos bits son 1, 0 en caso contrario.
- OR (|): realiza el “or” lógico entre dos valores. 0 si ambos bits son 0, 1 en caso contrario.
- XOR (^): realiza el “or” exclusivo lógico entre dos valores. 0 si ambos bits son iguales, 1 en caso contrario.

Bitwise Operators

- **Left Shift (<<):** Mueve todos los bits de un valor hacia la izquierda
- **Right Shift (>>):** Mueve todos los bits de un valor hacia la izquierda

Operador	Ejemplo	Resultado
NOT	! 0111	1000
AND	0101 & 0011	0001
OR	0101 | 0011	0111
XOR	0101 ^ 0011	0110
Left Shift	0100 << 1	1000
Right Shift	0100 >> 1	0010

Bitmasks (Visualizar)

- Un Integer (32/64 bit) se almacena en memoria como una secuencia de bits por lo que se pueden utilizar como un pequeño set de booleans marcando los bits.
- Todas las operaciones necesarios se pueden realizar con los operadores Bitwise
- Hacer esto es mucho mas eficiente que utilizar un array
- Representación:
  - Los bits son representados de derecha a izquierda, siendo el que esta más a la derecha el bit con la posición 0
  - EJ: para n = 11

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

Bitmasks (Visualizar)

- Multiplicar por 2:
  - Para esto se puede utilizar Left shift 1
  - Ej: 5 * 2 = 0101 << 1 = 1010 = 10
- Dividir por 2:
  - Para esto se puede utilizar Right Shift 1
  - Ej: 5 / 2 = 0101 >> 1 = 0010 = 2
  - Notar que se trunca el resultado, al igual que en la división de enteros.
- Marcar un bit (asignar 1)
  - Se utiliza OR
  - Para marcar el i-ésimo bit con 1: `n = n | (1 << i)`
  - Ej: si n = 0000, i = 2
    n = n | (1 << 2)
    n = 0100

Bitmasks (Visualizar)

- limpiar un bit (asignar 0)
  - Se utiliza AND y NOT
  - Para limpiar el i-ésimo bit con 0: **n = n & !(1 << i)**
  - Ej: si n = 0100, i = 2
    n = n & !(1 << 2)
    n = 0000
- Para revisar el valor de un bit
  - Se utiliza AND
  - Para revisar el bit i-ésimo: **n & (1 << i)**
  - Ej: si n = 0110, i = 2
    n & (1 << i) = true

Bitmasks (Visualizar)

- Para intercambiar el valor de un bit
  - Se utiliza XOR
  - Para intercambiar el bit i-ésimo: `n = n ^ (1 << i)`
  - Ej: si n = 0100, i = 1
    n = n ^ (1 << 1)
    n = 0110
- Obtener el valor del bit marcado menos significativo
  - El menos significativo es el primero desde la derecha
  - Se utiliza: `(n & -(n))`
- Marcar todos los bits
  - Para marcar todos los bits de un set con tamaño T con 1 se utiliza: `n = (1 << T) -1`
  - Tener cuidado con los overflow

Stack (Visualizar)

- Estructura de tipo LIFO (Last In First Out)
- Los valores son ingresados “arriba” y extraídos desde el mismo lugar
- **stack** en C++ y **Stack** en Java. En Python las listas se pueden utilizar como stack
- Operaciones
  - Inserción en $O(1)$
  - Eliminación en $O(1)$
  - Obtener valor en el tope en $O(1)$

Queue (Visualizar)

- Estructura de tipo FIFO (First In First Out)
- Los valores son ingresados en la parte posterior y extraídos desde el frente
- **queue** en C++ y **Queue** en Java. En Python las listas se pueden utilizar como queue
- Operaciones
  - Inserción en $O(1)$
  - Eliminación en $O(1)$
  - Obtener valor en el frente en $O(1)$

Double-ended Queue (Deque) (Visualizar)

- Mezcla los comportamientos de Stack y Queue
- **deque** en C++ y **Deque (ArrayDeque)** en Java. En Python las listas son prácticamente deque (tiene los métodos necesarios)
- Operaciones
  - Inserción en $O(1)$ en ambos lados
  - Eliminación en $O(1)$ en ambos lados
  - Obtener valor en el frente/tope en $O(1)$
  - Consultar un índice especifico en $O(1)$

Priority Queue (Visualizar)

- Es un tipo especial de estructura cuyo comportamiento se caracteriza por dar prioridad a los elementos, esto es, el elemento con mayor prioridad va a estar siempre en el Head de la estructura
- Usualmente es implementado utilizando un Heap binario
- **priority_queue** en C++ y **PriorityQueue** en Java. En Python utilizar una lista y la librería **heapq**
- Se utiliza en algoritmos de ordenamiento, grafos y otros
- Operaciones:
  - Inserción en $O(log_2 n)$
  - Extracción en $O(log_2 n)$

Binary Search Tree (BST) (Visualizar)

- Un árbol binario de búsqueda (BST) es una estructura que sigue la propiedad para cualquier nodo:
  - Los elementos en el subárbol izquierdo de este son menores a este
  - Los elementos en el subárbol derecho de este son mayores o iguales a este
- Operaciones de BST:
  - $O(\log_2 n)$ en búsqueda, insecion, eliminación en caso promedio. $O(n)$ en peor caso.
- Cuando el árbol está **balanceado** (el nodo raíz esta en el centro) se tiene que todas las operaciones anteriores tienen complejidad $O(\log_2 n)$
  - Ej de arboles balanceados: AVL, Red-Black Tree
- **map/set** en C++, **TreeMap/TreeSet** en Java son estructuras que implementan un Balanced BST
  - Si se usan estas implementaciones es difícil modificar o añadir información extra al árbol.

Hash Table (Visualizar)

- Estructura de tipo Key/value donde tanto la inserción y búsqueda de un valor especifico a través de su llave es ejecutado en $O(1)$
- Utiliza funciones de Hashing sobre la llave para determinar donde se almacena el valor.
- **unordered_map/unordered_set** en C++11, **HashMap/HashSet/HashTable** en Java, **Dictionary** en Python
- Desventaja: es propenso a colisiones si la función de hash no es suficientemente buena.
- Nota: Un simple array también puede ser usado como Hash Table con las condiciones adecuadas

Union Find Disjoint Sets (Visualizar)

- Es una estructura de datos para modelar una colección de conjuntos disjuntos con la habilidad de determinar de forma eficiente a que conjunto pertenece un elemento, conocer si dos elementos pertenecen al mismo conjunto y unir un par de conjuntos.
- Este tipo de operaciones no están bien soportadas por las librerías de los lenguajes de programación por lo tanto es necesario contar con una implementación propia
- Se utiliza una estructura en forma de árbol en que el representante es la raíz de este, de esta forma todos los elementos en un árbol determinado pertenecen a un mismo conjunto y son representados por su raíz (solo es necesario recorrer el árbol hasta la raíz)

Union Find Disjoint Sets (Visualizar)

- Importante notar que:
  - Se almacena una referencia al nodo padre en cada nodo.
  - La referencia al padre es actualizada cuando se determina el conjunto de un elemento para dejar al nodo raíz como el padre en cada nodo recorrido hacia la raíz.
  - Se almacena una variable “rango” (Rank) en cada nodo que denota la altura en el árbol
- Para unir un par de conjuntos basta con añadir un representante (el del menor rank) al otro representante (el de mayor rank)
- Operaciones en esta estructura tienen complejidad aproximada de $O(1)$

Union Find Disjoint Sets (Visualizar)

```
class UnionFind {                                              // OOP style
private: vi p, rank;                          // remember: vi is vector<int>
public:
  UnionFind(int N) { rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {                          // if from different set
      int x = findSet(i), y = findSet(j);
      if (rank[x] > rank[y]) p[y] = x;          // rank keeps the tree short
      else {                  p[x] = y;
                              if (rank[x] == rank[y]) rank[y]++; }
} } };
```

Implementación usando hash table aquí

Segment Tree (visualizar)

- Es una estructura para responder de forma eficiente consultas de rango dinámicas.
- Ej: Encontrar el elemento menor dentro de un arreglo entre los rangos [i , j]

La solución bruta es recorrer todos los elementos entre i y j, guardando el mínimo encontrado. ¿Cómo lo hacemos si este tipo de consultas se repiten muchas veces en un arreglo grande? Es necesario una mejor forma de resolverlo.

- SegmentTree corresponde a un árbol binario donde cada nodo corresponde a un rango. La raíz corresponde a [0, n-1].
- Si un nodo corresponde a los rangos [a , b], entonces el hijo izquierdo corresponde a [a, (a+b)/2] y el derecho a [(a+b)/2 + 1, b]

Segment Tree (visualizar)

- Además cada nodo tiene un valor, correspondiente al rango completo que representa. En caso del ejemplo anterior, el valor de un nodo cualquiera correspondería al mínimo entre los rangos que representa.
- Para consultar un rango especifico basta recorrer el árbol en dos direcciones hasta llegar a los limites exactos del rango consultado y finalmente efectuar una ultima operación (obtener el mínimo en el ejemplo) sobre cada valor obtenido
- Las operaciones tienen una complejidad de $O(log_2 n)$
- Es útil cuando hay actualizaciones frecuentes en el arreglo
- Existe una variante para también efectuar actualizaciones en un rango. Esta requiere una técnica llamada Lazy Propagation.

Segment Tree (visualizar)

```
class SegmentTree {          // the segment tree is stored like a heap array
private: vi st, A;             // recall that vi is: typedef vector<int> vi;
  int n;
  int left (int p) { return p << 1; }     // same as binary heap operations
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                                  // O(n)
    if (L == R)                          // as L == R, either one is fine
      st[p] = L;                                          // store the index
    else {                              // recursively compute the values
      build(left(p) , L               , (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R          );
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }
```

Segment Tree (visualizar)

```
int rmq(int p, int L, int R, int i, int j) {                    // O(log n)
  if (i >  R || j <  L) return -1; // current segment outside query range
  if (L >= i && R <= j) return st[p];                // inside query range

   // compute the min position in the left and right part of the interval
  int p1 = rmq(left(p) , L              , (L+R) / 2, i, j);
  int p2 = rmq(right(p), (L+R) / 2 + 1, R          , i, j);

  if (p1 == -1) return p2;    // if we try to access segment outside query
  if (p2 == -1) return p1;                                  // same as above
  return (A[p1] <= A[p2]) ? p1 : p2;                // as in build routine
}
```

SegmentTree con Lazy Propagation aquí

Binary Indexed (Fenwick) Tree (Visualizar)

- Otra estructura para resolver consultas por rango, aunque de forma más acotada, es el **Fenwick Tree** (Binary Indexed Tree – BIT). Particularmente sirve para **Range Sum Queries** (RSQ, consultas de suma por rango)
- Es utilizada para implementar tablas de frecuencia acumulada (Cumulative Frequency Table) dinámicas.

Arreglo	4	15	2	9	11
CFT	4	19	21	30	41

- Fenwick Tree Sirve en casos donde se requieren actualizaciones en la tabla. Para esto también se puede usar Segment Tree pero requiere mucho mas código.
- Utiliza operaciones a nivel de bits para la construcción $O(n\ log_2 n)$, actualizaciones $O(log_2 n)$ y consultas $O(log_2 n)$.

Binary Indexed (Fenwick) Tree (Visualizar)

```
class FenwickTree {
private: vi ft;
        // recall that vi is: typedef vector<int> vi;
public:
        FenwickTree(long n) { ft.assign(n + 1, 0); }
        // init n + 1 zeroes
        long LSOne(long n){return (n & (-n));}
        long long rsq(long b)
        {
                // returns RSQ(1, b)
                long long sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
                return sum;
        }
        long long rsq(long a, long b)
        {
                return rsq(b) - (a == 1 ? 0 : rsq(a - 1));
        }
        // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
        void adjust(long k, long v)
        {
                // note: n = ft.size() - 1
                for (; k < (long)ft.size(); k += LSOne(k)) ft[k] += v;
        }
};
```