ТЕОРИЯ И ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Лекция 4. Структуры данных. Управление памятью

Глухих Михаил Игоревич, к.т.н., доц.

mailto: glukhikh@mail.ru

Контейнеры

- Контейнер ~ “динамическое множество”
- Элемент = ключ (упорядоченный?) + данные
- NIL / NULL
- Основные операции
  - Универсальные
    - C.SEARCH(key)
    - C.INSERT(element)
    - C.DELETE(element)
  - Требующие сравнение
    - C.MINIMUM()
    - C.MAXIMUM()
    - C.SUCCESSOR(element)
    - C.PREDECESSOR(element)

Виды контейнеров

- Массивы
- Связанные списки
- Стеки, очереди, деки
- Хэш-таблицы
- Бинарные деревья

Виды контейнеров

- Массивы
  - SEARCH: O(N) / O(1), INSERT: O(N) / O(1), DELETE: O(N) / O(1)
- Связанные списки
- Стеки, очереди, деки
- Хэш-таблицы
- Бинарные деревья

Виды контейнеров

- Массивы
  - SEARCH: O(N) / O(1), INSERT: O(N) / O(1), DELETE: O(N) / O(1)
- Связанные списки
  - SEARCH: O(N), INSERT: O(N) / O(1), DELETE: O(N) / O(1)
- Стеки, очереди, деки
- Хэш-таблицы
- Бинарные деревья

Виды контейнеров

- Массивы
  - SEARCH: O(N) / O(1), INSERT: O(N) / O(1), DELETE: O(N) / O(1)
- Связанные списки
  - SEARCH: O(N), INSERT: O(N) / O(1), DELETE: O(N) / O(1)
- Стеки, очереди, деки
- Хэш-таблицы
  - SEARCH: O(1) / O(N), INSERT: O(1) / O(N), DELETE: O(1) / O(N)
- Бинарные деревья

Виды контейнеров

- Массивы
  - SEARCH: O(N) / O(1), INSERT: O(N) / O(1), DELETE: O(N) / O(1)
- Связанные списки
  - SEARCH: O(N), INSERT: O(N) / O(1), DELETE: O(N) / O(1)
- Стеки, очереди, деки
- Хэш-таблицы
  - SEARCH: O(1) / O(N), INSERT: O(1) / O(N), DELETE: O(1) / O(N)
- Бинарные деревья
  - SEARCH, INSERT, DELETE: O(logN) / O(N)

Динамическая память

- Есть определённый (большой) участок памяти
  - Удобно представим в виде массива
- Ожидается поддержка запросов вида (как минимум):
  - descriptor { size, … } = Alloc(size)
    - Can return NIL / NULL when “out of memory”
  - Free(descriptor)
- Суммарный размер памяти, выделенной всеми Alloc, не может быть больше размера исходного участка памяти

Упрощённый вариант

- Размер фиксирован и всегда равен 1 (байт, слов, ... = элементов массива)
- Реализация?

Упрощённый вариант

- Размер фиксирован и всегда равен 1 (байт, слов, ... = элементов массива)
- Реализация?
- Привязываем к каждому элементу массива признак «занято / свободно»
- Дескриптор = номер элемента
- Free: устанавливаем признак в «свободно»
- Alloc: находим первый свободный элемент и устанавливаем признак в «занято»
- См. пример
- Трудоёмкость, ресурсоёмкость?

Упрощённый вариант

- Размер фиксирован и всегда равен 1 (байт, слов, ... = элементов массива)
- Реализация?
- Привязываем к каждому элементу массива признак «занято / свободно»
- Дескриптор = номер элемента
- Free: устанавливаем признак в «свободно»
- Alloc: находим первый свободный элемент и устанавливаем признак в «занято»
- Трудоёмкость = O(1) Free / O(N) Alloc
- Ресурсоёмкость = O(N)

Упрощённый вариант

- Размер фиксирован и всегда равен 1 (байт, слов, ... = элементов массива)
- Трудоёмкость = O(1) Free / O(N) Alloc
- Ресурсоёмкость = O(N)
- Alloc: находим первый свободный элемент и устанавливаем признак в «занято»
- Реализация: а КАК находим первый свободный элемент?

Упрощённый вариант

- Размер фиксирован и всегда равен 1 (байт, слов, ... = элементов массива)
- Трудоёмкость = O(1) Free / Alloc
- Ресурсоёмкость = O(N)
- Alloc: находим первый свободный элемент и устанавливаем признак в «занято»
- Реализация: а КАК находим первый свободный элемент?
- Давайте вместо массива признаков «занято / свободно» хранить очередь свободных элементов
- Освобождающийся элемент помещается в хвост
- Размещающийся берётся из начала

Стандартный вариант

- Размер выделяемых участков не фиксирован и может меняться от 1 до полного размера кучи
- Какие дополнительные проблемы при этом возникают?

Стандартный вариант

- Размер выделяемых участков не фиксирован и может меняться от 1 до полного размера кучи
- Какие дополнительные проблемы при этом возникают?
  - Необходимо хранить не список свободных элементов, а список свободных блоков, и у каждого свободного блока может быть собственный размер

Стандартный вариант

- Размер выделяемых участков не фиксирован и может меняться от 1 до полного размера кучи
- Какие дополнительные проблемы при этом возникают?
  - Необходимо хранить не список свободных элементов, а список свободных блоков, и у каждого свободного блока может быть собственный размер
  - Фрагментация: свободные блоки могут «перемешаться» между занятыми, из-за чего будет не выделить участок памяти большого размера
  - И даже если такого перемешивания нет, обычно необходимо уметь находить соседние блоки и объединять их в один

Стандартный вариант

- Размер выделяемых участков не фиксирован и может меняться от 1 до полного размера кучи
- Известные алгоритмы
  - First Fit (Next Fit, Best Fit, Worst Fit)
  - Buddy System (алгоритм близнецов)
- Желательные характеристики
  - Трудоёмкость для Alloc / Free O(1)
  - Ресурсоёмкость не хуже O(N)

First / Next / Best / Worst Fit

- First Fit (первый подходящий)
  - При запросе, перебираем список свободных блоков и выдаём первый блок подходящего размера (то есть совпадающего с запрошенным или больше)
- Next Fit
  - Аналогично First Fit, но последний выбранный блок запоминается и при следующем запросе поиск начинается с него, а не с первого блока в списке (список закольцован)
- Best / Worst Fit (лучший / худший подходящий)
  - Выбирается блок достаточного размера с минимально / максимально возможным превышением размера над запрошенным
- Общее
  - При превышении размера, остаток либо не используется (если он небольшой), либо отрезается и объявляется отдельным блоком
  - Свободные блоки с соседними адресами могут объединяться в один

First / Next / Best / Worst Fit

- Варианты упорядочивания списка свободных блоков
  - По возрастанию адреса
  - По возрастанию размера (~ Best Fit)
  - По убыванию размера (~ Worst Fit)
  - По убыванию момента последнего использования
  - Отдельные списки для блоков одного (или близкого) размера

Buddy System

- Массив заранее разбивается на блоки определённого размера, обычно по степеням двойки
  - 4M = 2M + 1M + 512K + 256K +
    128K + 64K + 32K + 16K +
    8K + 4K + 2K + 1K + 512 + 256 + 256
- Список свободных блоков свой для каждой степени двойки
- При получении запроса его размер округляется до ближайшей степени двойки сверху
  - Если есть свободный блок этого размера – он и выдается
  - В противном случае разбивается на части блок большего размера (пример: надо 1K, есть 4K, 4K = 2K + 1K + **1K**)
- При освобождении соседние блоки могут объединяться

Анализ достоинств / недостатков

- Характеристики

Анализ достоинств / недостатков

- Характеристики
  - Трудоёмкость
  - Ресурсоёмкость
  - Количество потерянной памяти
  - Фрагментация

Worst Fit

- Блоки храним по убыванию размера
- Большие блоки разбиваем
- Характеристики
  - Трудоёмкость: Alloc O(1), Free O(lgN)
  - Ресурсоёмкость: O(N)
  - Количество потерянной памяти: минимально
  - Фрагментация: алгоритм имеет трудности при выделении блоков большого размера

Best Fit

- Блоки храним по возрастанию размера
- Большие блоки разбиваем
- Характеристики
  - Трудоёмкость: Alloc O(lgN), Free O(lgN)
  - Ресурсоёмкость: O(N)
  - Количество потерянной памяти: минимально
  - Фрагментация: алгоритм имеет тенденцию создавать блоки очень маленького размера

Buddy System

- Отдельные списки для блоков одного размера
- Разбиваем исключительно по степеням двойки
- Характеристики
  - Трудоёмкость: Alloc O(1), Free O(1)
  - Ресурсоёмкость: O(lgN)
  - Количество потерянной памяти: около 25%
  - Фрагментация: минимальна