Суффиксные деревья

- Грани и *Z*-блоки – 2 известные схемы препроцессинга строк
- Суффиксные деревья – структура для индексирования строк (П. Вайнер – 1973)
- Глубже выявляет архитектуру строки
- Имеет много приложений в задачах обработки строк
- Поиск точных совпадений образца за линейное время – не менее эффективно, чем рассмотренные ранее алгоритмы
- Решение за линейное время многих других задач на строках, более сложных, чем точные совпадения
- Суффиксное дерево – один из наиболее мощных современных инструментов вычислений на строках

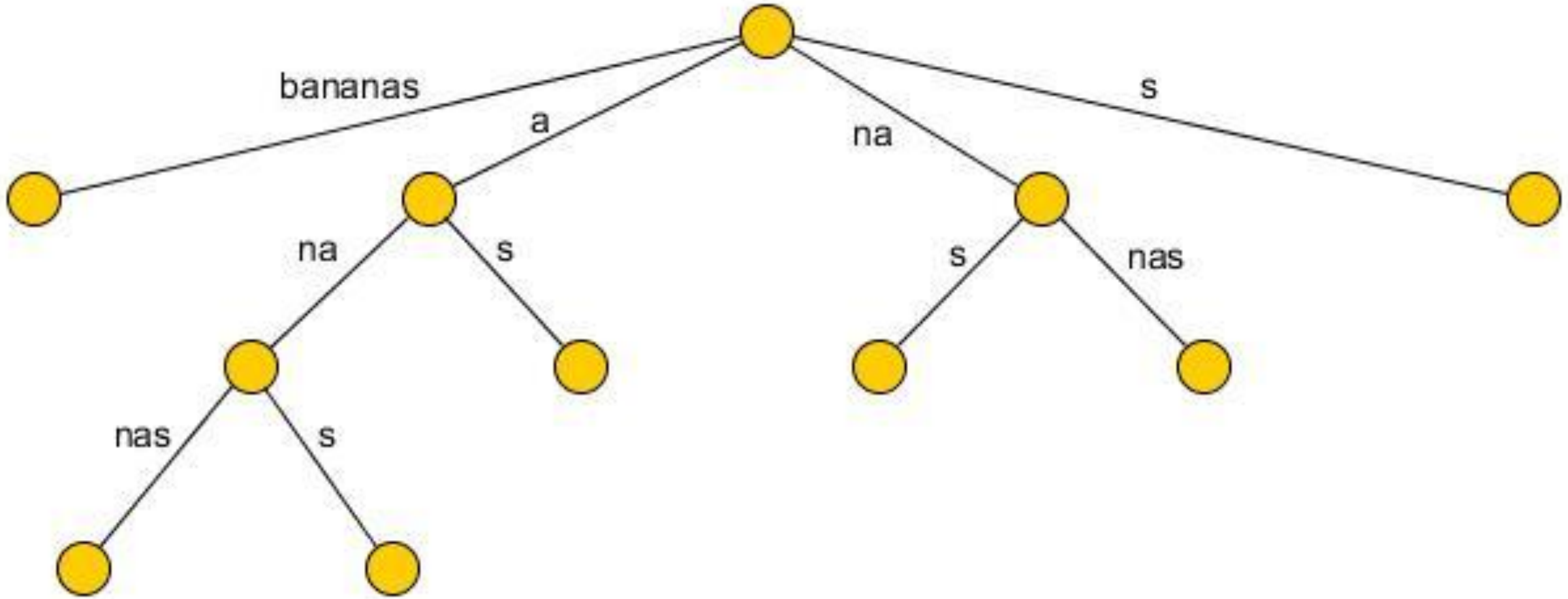
Построение и применение

- Задача эффективного построения суффиксного дерева нетривиальна
- П. Вайнер показал, что она решается за линейное время
- Это решение Д. Кнут назвал «алгоритмом 1973 года»
- Классическое приложение: поиск всех вхождений образца P длины m в текст T длины n
- За время $O(n)$ строится соответствующее тексту суффиксное дерево
- На основе дерева поиск в тексте T любого образца P длины m может быть выполнен за время $O(m)$
- Без препроцессинга образца и независимо от длины самого текста!
- Формально будем использовать динамическую память, низкоуровневая оптимизация остается в качестве следующего шага

Основное определение

- Используем минимальную терминологию
- *Определение*. Дана строка *S* длины *n*. Ее суффиксным деревом называется ориентированное дерево, обладающее свойствами:
  1) ∀ дуга помечена непустой подстрокой *S*;
  2) ∀ внутренняя вершина содержит не менее двух дочерних вершин;
  3) метки ∀ пары дуг из общей вершины начинаются с различных символов;
  4) имеет *n* листьев (= общему количеству суффиксов);
  5) ∀ путь от корня до листа «собирает» некоторый суффикс строки *S*.
- *Неформально*: взять все суффиксы строки *S*, подвесить их в корне будущего дерева и общие части объединить.

Пример: строка *bananas*



Проблема существования

- Из определения суффиксного дерева не следует его существование для произвольной строки
- Пример: *banana*
- Рассмотрим два суффикса: *anana* и *ana*.
- Особенность: второй представляет собой префикс первого
- По условиям 4)–5) определения каждому из них в дереве должен соответствовать собственный путь от корня до листа
- Но по условию 3) их пути не могут «разойтись» ни в какой вершине
- Для слова *banana* построить суффиксное дерево невозможно

Дополнения к определению

- Исследуемый текст дополняется в конце символом, не относящимся к исходному алфавиту
- Нередко «терминальный символ» обозначают $
- Тогда ни один суффикс не будет собственным префиксом другого
- Существование суффиксного дерева гарантировано, и все его полезные свойства сохраняются
- Суффикс из одного терминального символа обычно не рассматривается; тогда дерево для модифицированного слова также содержит *n* листьев
- Листья нередко помечаются соответствующими суффиксами (или индексами в *S* их начальных символов)

Пример: суффиксы строки *abracadabra*$

S[0..] = *abracadabra*$

S[1..] = *bracadabra*$

S[2..] = *racadabra*$

S[3..] = *acadabra*$

S[4..] = *cadabra*$

S[5..] = *adabra*$

S[6..] = *dabra*$

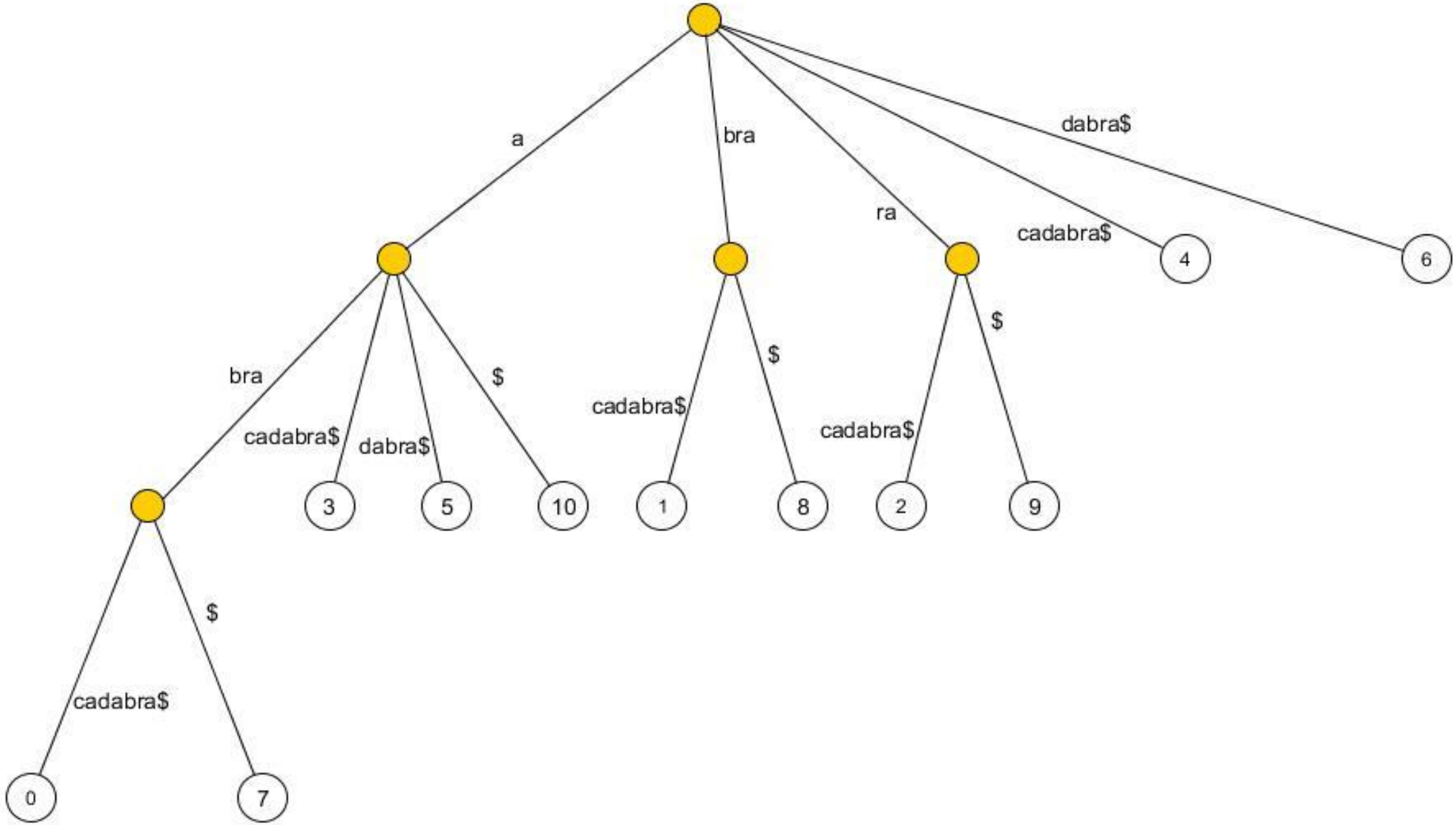
S[7..] = *abra*$

S[8..] = *bra*$

S[9..] = *ra*$

S[10..] = *a*$

Пример: дерево строки *abracadabra*$



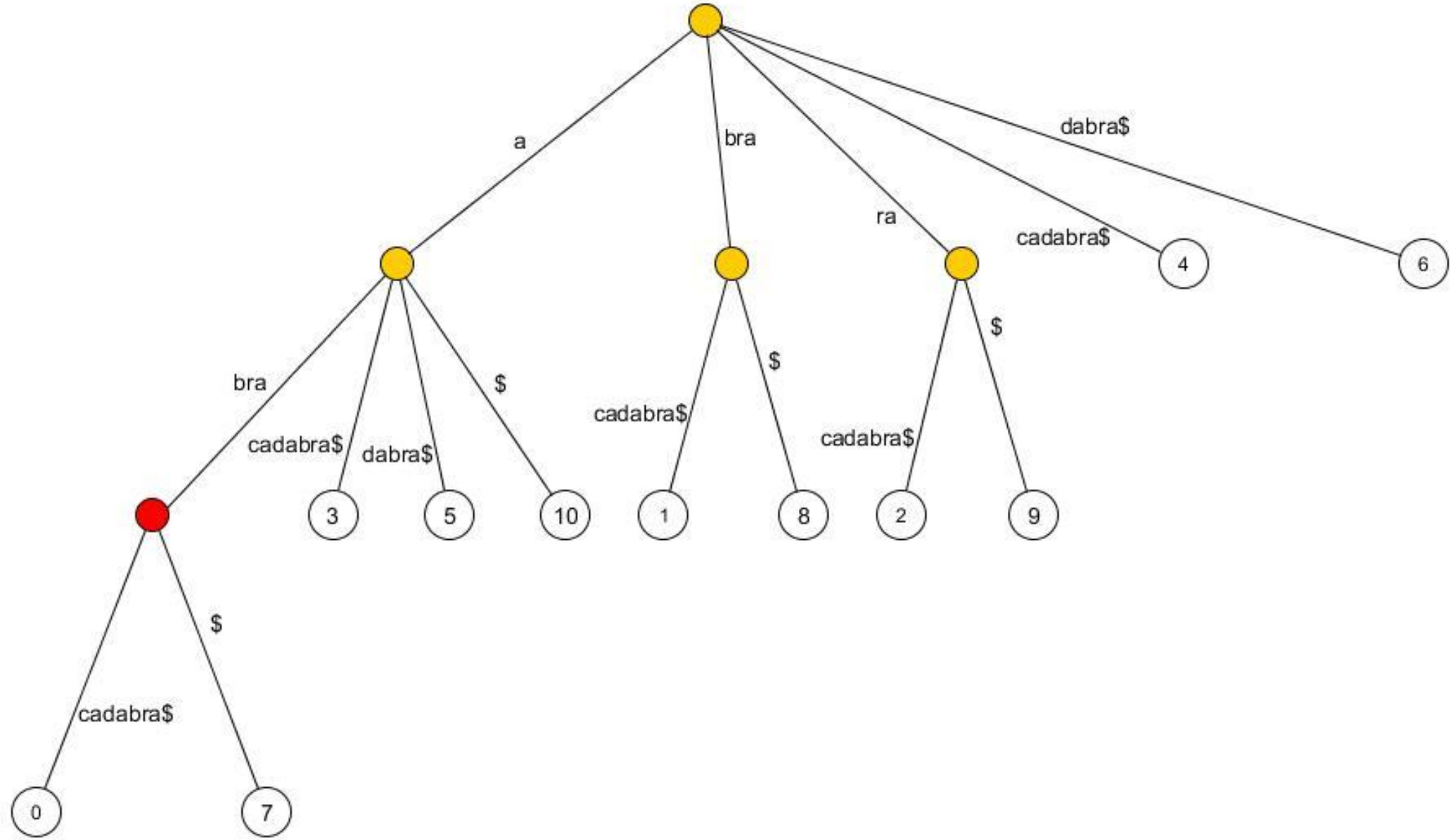
Поиск на суффиксном дереве – движение

- Даны текст *T* длины *n* и его суффиксное дерево
- Требуется обнаружить все вхождения в *T* образца *P* длины *m*
- Начинаем движение от корня дерева, «прикладывая» образец к наиболее благоприятному пути
- Точнее: двигаемся в направлении от корня, пока символы образца последовательно совпадают с символами меток некоторого пути
- В силу условия 3) этот путь определяется единственным образом
- Прекращение движения возможно в двух случаях:
  1) несовпадение очередного символа из *P*;
  2) образец *P* исчерпан (совпадет с набором меток некоторого пути или его части)

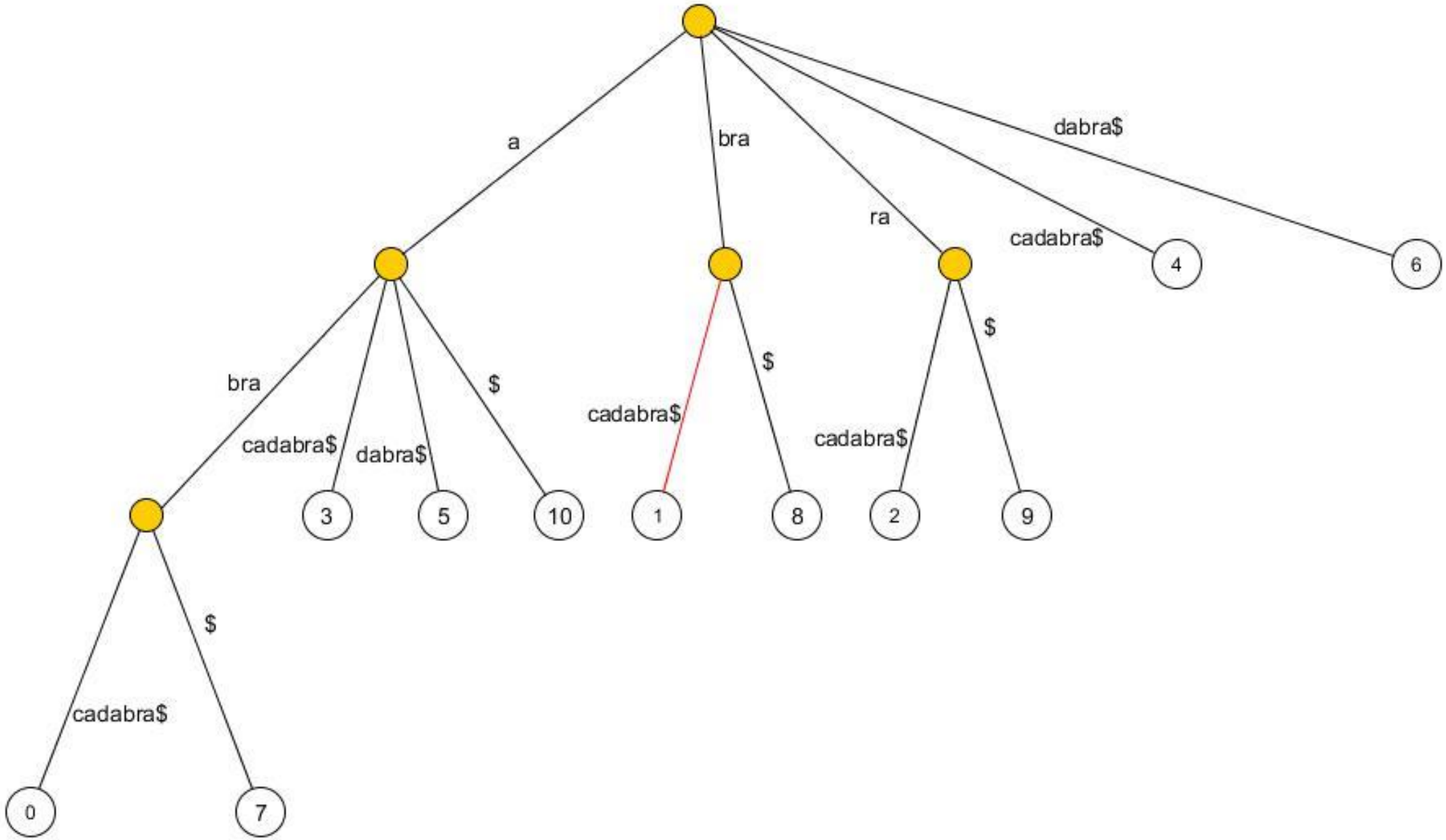
Поиск на суффиксном дереве – результаты

- Прекращение движения возможно в двух случаях:
  1) несовпадение очередного символа из *P*;
  2) образец *P* исчерпан (совпадет с набором меток некоторого пути или его части)
- Первый вариант означает, что вхождений *P* в *T* нет
- Во втором все вхождения образца соответствуют листьям дерева ниже точки совпадения последнего символа *P*
- Точнее: позиции вхождений – это метки-индексы всех таких листьев

Пример: поиск подстроки *abra*



Пример: поиск подстроки *braca*



Сложность поиска на суффиксном дереве

- Алфавит конечен: вычисление в ∀ вершине дерева (поиск продолжения пути) занимает константное время
- В результате сложность поиска совпадения *P* с набором меток пути пропорциональна длине образца *m*
- Далее требуются затраты на перечисление дочерних листьев для позиции, где остановилось движение
- Обход поддерева (например, в глубину) занимает линейное время по числу его вершин
- Дерево не менее чем бинарное (условие 2)) ⇒ общее число вершин не превосходит удвоенного числа листьев
- Сложность перечисления листьев пропорциональна их количеству *k*
- *Итого*: сложность поиска на суффиксном дереве оценивается как $O(m + k)$, где *k* – искомое число вхождений образца в текст

Вопросы реализации поиска на дереве

- Чтобы при прохождении вершины время работы не зависело от длины алфавита, достаточно исходящие дуги хранить в виде массива, индексированного по символам алфавита
- Возможны другие способы хранения множества дуг – линейный список, сбалансированное дерево, хеширование и т.п.
- Тогда размер алфавита сказывается на времени обработки, но более экономно используется память
- При использовании массива, для ускорения второй стадии поиска – обхода дочерних листьев – непустые элементы можно связать дополнительным списком
- Ниже потребуются структуры, связанные с суффиксным деревом
- Некоторые типы данных играют существенную роль, поэтому приводится их описание

Структуры реализации суффиксного дерева

```
// Дуга
typedef  struct
{
  int  iBeg,  iEnd;  //  Индексы символов метки (в исходной строке)
  PNode  pDestVert; // Вершина, куда входит дуга (для листа = NULL)
  int  iDestVert;  // Индекс листа, куда входит дуга (для внутренней = -1)
}  Arc, *PArc;
// Вершина дерева
typedef  struct
{
  PArc  arcs[nAlpha];   //  Массив ссылок на исходящие дуги
}  Node, *PNode;        //  nAlpha – длина алфавита
```

Оценка памяти

- Пусть текст имеет длину *n*, тогда и общее число его суффиксов составляет *n*
- По определению суффиксное дерево, будучи не менее чем бинарным, имеет ровно 1 лист для каждого суффикса, т.е. всего *n*
- Тогда, согласно свойству бинарного дерева, общее число его вершин не превосходит 2*n*, соответственно число дуг – 2*n*-1
- Таким образом, при описанном выше формате данных необходимый для хранения суффиксного дерева объем памяти оценивается как *O*(*n* x *nAlpha*)
- При использовании списков дуг эту оценку можно снизить до *O*(*n*)

Поиск подстроки по меткам дуг – инициализация

- Возвращает ссылку на дугу, на которой остановилось движение
- Еще результаты – два параметра по ссылке:

  idxSubstr – индекс первого несовпавшего символа подстроки (образца) и

  idxArc – индекс первого несовпавшего символа метки дуги

```
Find-SuffixTree-Arc (str, substr, m, PNode pTree, &idxSubstr, &idxArc)
{
  PArc  pArc = NULL; // Дуга, на которой остановится поиск
  idxSubstr =  idxArc = 0;  // Индексы несовпавших символов
  PNode  pCurrNode = pTree;  //  Начинаем движение от корня
  bStopped = 0;
  while  (!bStopped && pCurrNode)
```

Поиск подстроки по меткам дуг – вычисления

```
  {
    PArc  pNextArc  =  pCurrNode->arcs[substr[idxSubstr]];
    if  (pNextArc)  {  // Есть совпадение с начальным символом метки дуги
      pArc  =  pNextArc; idxArc = pArc->iBeg;
      // Сравниваем последующие символы
      while  (++idxSubstr < m && ++idxArc < pArc->iEnd + 1
                                  && substr[idxSubstr] == str[idxArc]);
      if  (idxArc <= pArc->iEnd)  bStopped = 1;  // Не прошли метку
      else  pCurrNode = pArc->pDestVert;   //  Переход к следующей вершине
    }
    else bStopped = 1; //  Нет продолжения пути
  }
if  (idxSubstr == m) ++idxArc; //  Чтобы idxArc было за границей совпадения
return  pArc; }
```

Обход дочерних листьев

```
ST-Leaves-Traversal (PArc pStartArc,  nAlpha)
{  //  pStartArc – стартовая дуга обхода; nAlpha – длина алфавита
   if  (pStartArc->iDestVert >= 0)  // Если дуга направлена к листу
      printf ("Найдена позиция %d\n", pStartArc->iDestVert);
   else
   {  //  Дуга направлена к внутренней вершине дерева
      PNode pStartNode = pStartArc->pDestVert;
      for  (k = 0; k < nAlpha; ++k)
      {  //  Перебор дуг дочерней вершины
         PArc  pArc = pStartNode->arcs[k];
         if  (pArc)  ST-Leaves-Traversal (pArc,  nAlpha);
      }
}}
```

Построение суффиксного дерева

- Интуитивно возникающий алгоритм неэффективен, но помогает глубже понять архитектуру суффиксного дерева
- Простая идея из классической задачи построения дерева поиска по множеству ключей – «найти ключ или добавить его»
- Процесс поиска очередного несуществующего в дереве ключа приводит в позицию его добавления
- При построении суффиксного дерева роль «ключей» играют суффиксы исходной строки
- Начать с простой дуги, помеченной наибольшим суффиксом 0, то есть $S[0..]\$$
- Далее последовательно добавлять суффиксы $S[i..]\$$, $i = 1, \ldots, n-1$

Описание наивного алгоритма: *i*-я итерация

- *S*[*i*..]$ ищем в построенном дереве (алгоритм поиска описан ранее)
- Ввиду уникальности суффиксов движение остановится до исчерпания *S*[*i*..]$
- Остановка – внутри некоторой дуги (*u*, *v*) или в вершине *w* = *v* (не листе!)
- *S*[*k*..]$ – часть добавляемого суффикса, которая не нашлась в дереве, *k* ≥ *i*
- Пусть поиск остановился внутри дуги (*u*, *v*), имеющей некоторую метку *s*:
  - разобьем эту дугу на две (*u*, *w*) и (*w*, *v*)
  - первой из них припишем совпавшую часть метки *s* (она завершается символом *S*[*k*-1]), второй – не совпавшую
- Выведем из вершины *w* дугу (*w*, *i*) с меткой *S*[*k*..]$, завершающуюся новым листом дерева с индексом *i*
- Символ *S*[*k*] не был найден в процессе поиска, поэтому метки всех дуг, выходящих из *w*, будут начинаться с различных символов

Замечания к наивному алгоритму

- Перечисление и добавление суффиксов может быть произведено в любом другом порядке, например, при $i = n-1, n-2, \ldots, 0$
- Алгоритм обрабатывает *n-1* суффикс $S[i..]\$$, $i = 1, \ldots, n-1$
- Каждый из суффиксов в худшем случае требует порядка *n-i* операций сравнения символов
- Таким образом, сложность алгоритма оценивается как $O(n^2)$
- При его реализации оказывается полезной приведенная ранее функция *Find-SuffixTree-Arc*
- Ниже для читабельности формально используется динамическая память, однако на практике рекомендуются массивы структур

Вспомогательная функция *ST-Vert-Init*

```
//  Создание вершины дерева
ST-Vert-Init ()
{
   PNode pVert = (PNode) calloc (1, sizeof (Node) );  // Заполн. нулями
   return  pVert;
}
```

Вспомогательная функция *ST-Arc-Init*

```
//  Создание исходящей дуги в вершине дерева
ST-Arc-Init (PNode pSNode, chArcIdx, iBeg, iEnd, pDestVert, iDestVert)
{  // chArcIdx – код начального символа метки дуги
   PArc pArc = (PArc) calloc (1, sizeof (Arc) );  //  Заполн. нулями
   pArc->iBeg = iBeg;  pArc->iEnd = iEnd; pSNode->arcs[chArcIdx] = pArc;
   pArc->pDestVert = pDestVert; pArc->iDestVert = iDestVert;
   return  pArc;
}
```

Наивный алгоритм – инициализация

- Возвращает ссылку на корневую вершину созданного дерева

ST-Buid-Naive (str)

```
{
   PNode  pTree = NULL; n = strlen(str);
   // Корень дерева и его начальная дуга
   pTree = ST-Vert-Init ();
   ST-Arc-Init (pTree, str[0], 0, n - 1, NULL, 0);
```

Наивный алгоритм – поиск суффикса

```
for  (i = 1;  i < n - 1; ++i)
{
   // "Поиск" очередного суффикса на дереве
   PArc pUVArc = Find-SuffixTree-Arc (str, &str[i], n-i, pTree, idxSuff, idxArc);
   PNode pWNode; // Вершина-источник дуги для нового суффикса
   if  (!pUVArc)  pWNode = pTree; //  Поиск остановился в корне
```

Наивный алгоритм – создание вершины

```
else  if (idxArc  <=  pUVArc->iEnd)
{ // Поиск остановился внутри дуги (U, V), требуется ее разделение
   pWNode = ST-Vert-Init ();  // Новая разделяющая вершина
   PArc  pWVArc = ST-Arc-Init (pWNode, str[idxArc], idxArc, pUVArc->iEnd,
                               pUVArc->pDestVert,  pUVArc->iDestVert); // Дуга из W в V
   pUVArc->pDestVert = pWNode; pUVArc->iEnd = idxArc - 1; // Дуга из U в W
   pUVArc->iDestVert = -1;
}
```

Наивный алгоритм – окончание

```
    // Поиск остановился в конце дуги (U, V)
    else  pWNode = pUVArc->pDestVert;
    // Добавить новую дугу из вершины W в лист
    ST-Arc-Init (pWNode, str[i + idxSuff], i + idxSuff,  n - 1,  NULL, i);
  }
  return  pTree;
}
```